Technische Universität Ilmenau
Institut für Mathematik

**Preprint No. M 99/23**

# MATLAB - Teil II, Speicheraspekte, spezielle LGS, SVD, EWP, Graphik, NLG, NLGS

Neundorf, Werner

**1999**

**Impressum:**
Hrsg.: Leiter des Instituts für Mathematik
Weimarer Straße 25
98693 Ilmenau
Tel.: +49 3677 69 3621
Fax: +49 3677 69 3270
http://www.tu-ilmenau.de/ifm/

ISSN xxxx-xxxx

ilmedia

**Zusammenfassung**

This is a tutorial on teaching and programming in MATLAB.
It is based on scripts and exercises in the course of numerical mathematics for students of the faculties Electrical Engineering and Information Technology and Computer Science and Automation after first term.
The part I contains basic aspects and elements of numerical linear algebra, especially methods for systems of linear equations. This second part gives some hints about storage and import/export of data files, furthermore MATLAB-based algorithms and programming tools for special systems of linear equations, for eigenvalue problems and singular value decomposition, graphic aspects, nonlinear equations and systems of equations. Further part concerns polynomial and spline interpolations, approximation and a few integration.

## Vorwort

MATLAB is an interactive, matrix-based system for scientific and engineering calculations. You can solve complex numerical problems without actually writing a program. The name MATLAB is an abbreviation for MATrix LABoratory.

A few words to those who are familiar with other programming languages.

- MATLAB is a user-friendly high-level programming language and important technical computing environment. It also includes a number of functional programming constructs, modeling, simulation and prototyping, application development and design.

- MATLAB has a rich environment of powerful toolboxes and data visualization. It offers programming structures like control flows, selections, decisions and $m$-files functions.

- Students can affordably use this powerful numeric computation, data analysis and visualization software in their undergraduate and graduate studies. The student edition encapsulates a wide range of disciplines.

- MATLAB is not strongly typed like C and Pascal. No declarations are required. It is more like Basic and Lisp in this respect. You can dynamically link C or FORTRAN subroutines. Some type checking is done at run time.

- MATLAB suitable for running numerically intensive programs with double-precision numerical calculations. On the other site there are, based on Maple V, many symbolic tools and symbolic functions to combine, simplify, differentiate, integrate, and solve algebraic and differential equations. The symbolic toolbox and the subroutines concept of MATLAB seems to be not so efficiently as is described in Maple.

- MATLAB is available for a number of environments: Sun/Apollo/VAXstation/HP workstations, VAX, MicroVAX, Gould, PC, Apple Macintosh, and several parallel machines.

The aim is to show how you can write simple instructions, commands or programs in MATLAB for doing numerical calculations, linear algebra, and programs for simplifying or transforming expressions, equations, mathematical formulas or arrays.

It is assumed that the reader is familiar with using MATLAB interactively. For beginners we propose the introductional tutorial, the so called Primer. The purpose of this Primer based on the version 3.5 is to help you begin to use MATLAB. They can best be used hands-on. You are encouraged to work at the computer as you read the Primer and freely experiment with examples.

This document is based on **MATLAB version 4.2c1**.
MATLAB development continues. New versions come out every one or two years which contain not only changes to the mathematical capabilities of MATLAB, but also changes to the programming language and user interface. The MATLAB 5.2 and 5.3 highlights you can find on the web site

`http://www.mathworks.com/products/matlab/highlights.shtml` .

We are pleased and somewhat surprised to see how quickly this movement is already happening. For this reason, I have applied constructs in the language that will be probable in the language in future versions of MATLAB.

You should liberally use the on-line help facility for more detailed information. After entering MATLAB the command `help` will display a list of functions for which on-line help is available. The command `help` *functionname* will give information about a specific function. You can preview some of the features of MATLAB by entering the command `demo`. Even better, access help from the menu.
In the bibliography there are given some useful sources of information and supplemental workbooks for MATLAB.

# Inhaltsverzeichnis

# 1 Speicheraspekte und Datenfiles

## 1.1 Felder und ihre Größe

Wir definieren zunächst einige Variablen, Vektoren und Matrizen.
Als Ausgabe notieren wir nur wenige ausgewählte Variablen.

```
b = [-1 -2];
c = [-3 -4]';
x = 1e-3;
a11 = 0.1; a12 = 12; a21 = 34; a22 = 100;
A = [a11 a12; a21 a22];
B = [a11+a12 a11*a21 ; a22^2  10-a22]

B =
  1.0e+004 *
    0.0012   0.0003
    1.0000  -0.0090

m = 3; n = 5;
I = eye(m)
Imn = eye(m,n)
RAND100 = rand(100);   % grosse Zufallszahlenmatrix
l = length(RAND100)
RAND5 = rand(5)
A1 = [1:4; 5:8; 9:12; 13:16]
C = [A1; A B]          % strukturierte Matrix

C =
  1.0e+004 *
    0.0001   0.0002   0.0003   0.0004
    0.0005   0.0006   0.0007   0.0008
    0.0009   0.0010   0.0011   0.0012
    0.0013   0.0014   0.0015   0.0016
    0.0000   0.0012   0.0012   0.0003
    0.0034   0.0100   1.0000  -0.0090
```

Mit den Kommandos `who, whos` betrachten wir nun die aktuell gespeicherten Variablen als alphabetisch geordnete Liste mit möglichen Zusatzinformationen.

```
who  % variables currently in memory

Your variables are:
A         I         a11       b         n
A1        Imn       a12       c         x
B         RAND100   a21       l
C         RAND5     a22       m
```

```
whos  % more extensive
       Name        Size       Elements     Bytes    Density   Complex

          A       2 by 2              4        32       Full      No
         A1       4 by 4             16       128       Full      No
          B       2 by 2              4        32       Full      No
          C       6 by 4             24       192       Full      No
          I       3 by 3              9        72       Full      No
        Imn       3 by 5             15       120       Full      No
    RAND100     100 by 100        10000     80000       Full      No
      RAND5       5 by 5             25       200       Full      No
        a11       1 by 1              1         8       Full      No
        a12       1 by 1              1         8       Full      No
        a21       1 by 1              1         8       Full      No
        a22       1 by 1              1         8       Full      No
          b       1 by 2              2        16       Full      No
          c       2 by 1              2        16       Full      No
          l       1 by 1              1         8       Full      No
          m       1 by 1              1         8       Full      No
          n       1 by 1              1         8       Full      No
          x       1 by 1              1         8       Full      No

Grand total is 10109 elements using 80872 bytes
```

## 1.2 Protokoll

Von der Session oder ausgewählten Teilen kann man mittels des Kommandos `diary` ein Protokoll im Textformat (ASCII-Datei) anfertigen, ev. zwecks späterer Einbindung in andere Dokumente.

```
% Oeffnen einer neuen Protokolldatei bzw. Anhaengen an eine vorhandene
%
diary stor1.txt
% stor1.txt - Beginn der Protokolldatei
%
diary on   % kann entfallen, da automatisch
a11
sin(1)^2+cos(1)^2
diary off  % auch diary (wie Schalter), mit Umschalten Uebernahme in Datei
% diese Zeile und nachfolgendes werden nicht in Datei uebertragen
A
A1
diary on   % auch diary, Umschalten
% Fortsetzung des Protokolls mit dieser Zeile
a12
diary off  % am besten Protokoll damit beenden
```

In der Protokolldatei *stor1.txt* befinden sich dann folgende Einträge.

```
% stor1.txt -  Beginn der Protokolldatei
%
diary on  % kann entfallen, da automatisch
a11

a11 =
    0.1000

sin(1)^2+cos(1)^2

ans =
     1

diary off  % auch diary (wie Schalter), mit Umschalten Uebernahme in Datei
% Fortsetzung des Protokolls mit dieser Zeile
a12

a12 =
    12

diary off  % am besten Protokoll damit beenden
```

## 1.3 Speichern von Variablen

Während der Ausführung von Kommandos kann man einige oder alle Variablen aus dem Arbeitsspeicher (Workspace) in ein File übertragen. Das kann im binären MAT-File-Format erfolgen oder ein ASCII-Datenfile mit der Option `-ascii` bzw. `/ascii` sein. Zu letzterem sind auch noch weitere Optionen möglich wie `-double` oder `-tabs`.
Entsprechend können die Daten auch wieder geladen werden.
Dazu gibt es die Kommandos `save` und `load`.

Die **Grundvariante** ist wie folgt: Speichere alle Variablen im Arbeitsspeicher auf dem Standard-Binärfile *matlab.mat* im aktuellen Verzeichnis und lade diese dann wieder. Dabei kann zu Testzwecken zwischendurch eine Kontrolle der Belegung der Variablen erfolgen.
Beim Abspeichern werden auch die Variablennamen gemerkt.

```
save

Saving to: matlab.mat

% clear all variables
clear all
% control
```

```
A

A =
   []

% load saved variables
load

Loading from: matlab.mat

% control
A

A =
    0.1000   12.0000
   34.0000  100.0000
```

Durch zusätzliche Parameter und Optionen lassen sich die Basisversionen nunmehr erweitern. Es sollen hier nur einige davon notiert werden.

**Arbeit mit Binärfiles**

- Speichern mit Dateinamen

```
% save all variables in the binary file  stor1.mat
save stor1     % bzw.  save stor1.mat
clear all
% control
A
% load saved variables
load stor1     % bzw.  load stor1.mat
% control
A
```

- Speichern einer Variablen

```
% save a variable (here a matrix) in the binary file  stor11.mat
save stor11 A  % bzw.  save stor11.mat A
clear A
% control
A
% load saved variable
load stor11
% control
A
```

- Speichern von verschiedenen Variablen

```
% save some variables in the binary file  stor12.mat
save stor12 A B c l
clear A B c l
A, B, c, l
% load saved variables
load stor12
% control
A, B, c, l
```

**Arbeit mit ASCII-Files**

Dabei ist die Darstellung flexibel, indem man Lang- oder Kurzformen benutzen kann.

- Langform zum Speichern einer einfachen oder Feld-Variablen

```
% save a variable or array variable in the ASCII file  stor13.mat
save stor13.dat A -ascii   % bzw.  save stor13.dat A /ascii
clear A
A
% load the saved variable
load stor13.dat
% control
A
stor13
A = stor13
```

Das `save`-Kommando erzeugt zunächst die Textdatei (ASCII-File) *stor*13.*dat*.

```
1.0000000e-001  1.2000000e+001
3.4000000e+001  1.0000000e+002
```

aus deren Inhalt nicht der Name der abgespeicherten Matrix zu erkennen ist. Das Kommando `load stor13.dat` kann nur dieses rechteckige Datenfeld (Matrix) lesen und unter der Bezeichnung *stor*13 ablegen. Danach erfolgt gegebenenfalls wie oben die Umbenennung auf den Matrixnamen.

- Kürzere Form zum Speichern einer Matrix (Matrixname=Dateiname)

```
% save an array variable A in the ASCII file  A.dat
save A.dat A -ascii
clear A
A
% load the saved variable
load A.dat
% control
A
```

- Kurzform zum Speichern einer Matrix mit Dateinamen ohne Extension

```
% save an array variable A in the ASCII file  A
save A A -ascii
clear A
A
% load saved variables,
load A -ascii  % nicht nur  load A
% control
A
```

  Beim Laden muß die Option `-ascii` stehen, da sonst die Datei *A.mat* gesucht wird.

- Speichern mehrerer einfacher Variablen

```
% save some variables in the ASCII file  stor14.dat
save stor14.dat m n a11 x -ascii
clear m n a11 x
% load saved variables
load stor14.dat
% control
m, n, a11, x
stor14
% Zerlegung des Spaltenvektors stor14(1..4,1..1)
m = stor14(1,1)
n = stor14(2,1)
a11 = stor14(3,1)
x = stor14(4,1)
```

- Speichern mehrerer Felder mit jeweils gleicher Spaltenanzahl

```
% save some array variables in the ASCII file  stor15.dat
% have the same number of column
save stor15.dat A B b -ascii
clear A B b
% load saved variables
load stor15.dat
% control
A, B, b
stor15

stor15 =
  1.0e+004 *
    0.0000    0.0012
    0.0034    0.0100
```

```
    0.0012    0.0003
    1.0000   -0.0090
   -0.0001   -0.0002

% Zerlegung des Rechtecktableaus stor15(1..5,1..2),
% um die Matrizen zu erhalten
A = stor15(1:2,1:2)
B = stor15(3:4,1:2)
b = stor15(5,1:2)

A =
    0.1000   12.0000
   34.0000  100.0000

B =
  1.0e+004 *
    0.0012    0.0003
    1.0000   -0.0090

b =
    -1    -2
```

- Speichern mehrerer Felder mit unterschiedlicher Spaltenanzahl

```
% save some different array variables in the ASCII file  stor16.dat
save stor16.dat A B b x c I -ascii
```

Die Textdatei *stor*16.*dat* hat folgende Belegung.

```
 1.0000000e-001  1.2000000e+001
 3.4000000e+001  1.0000000e+002
 1.2100000e+001  3.4000000e+000
 1.0000000e+004 -9.0000000e+001
-1.0000000e+000 -2.0000000e+000
 1.0000000e-003
-3.0000000e+000
-4.0000000e+000
 1.0000000e+000  0.0000000e+000  0.0000000e+000
 0.0000000e+000  1.0000000e+000  0.0000000e+000
 0.0000000e+000  0.0000000e+000  1.0000000e+000
```

Das `load`-Kommando kann nicht verwendet werden wegen der unterschiedlichen Spaltenanzahl der im File gespeicherten Variablen.
Entweder müssen vor Abspeichern die Variablen in entsprechende verträgliche Felder "eingebettet" werden, oder man kann natürlich auch die Datei *stor*16.*dat* editieren und zum Laden passend machen.

# 2 Spezielle LGS, Pseudoinverse, Singulärwertzerlegung

Wir betrachten insbesondere LGS mit singulärer Koeffizientenmatrix, wo keine oder unendlich viele Lösungen auftreten.
Dabei ist zu beachten, daß wegen der Anwendung zumeist numerischer Algorithmen in MATLAB die Verifikation der Singularität "verschmiert" sein kann und diesbezügliche Warnungen wie

*Warning: Matrix is close to singular or badly scaled.*
*Results may be inaccurate.*

*Warning: Matrix is singular to working precision.*

*Warning: Matrix is rank deficient; solution does not exist.*
*Warning: Matrix is rank deficient; solution is not unique.*

sehr ernst zu nehmen sind.

Natürlich gibt es Möglichkeiten, solche speziellen Eigenschaften mehr oder weniger genau zu erkennen, bei kleineren Systemen eher als bei großen.

## 2.1 Spezielle LGS

In [9] sind die Grundlagen für die Anwendung der Kommandos `inv, rref, lu` beschrieben und an zahlreichen Beispielen illustriert.
Die built-in-Funktion `rref` implementiert den Gaußalgorithmus als Row reduce Variante einer Rechteckmatrix auf die Gauß-Jordan-Form. Ihre Anwendung kann zur Rangbestimmung genutzt werden. Besondere Argumente sind die Koeffizientenmatrix eines LGS oder die erweiterte Koeffizientenmatrix. Ist die Koeffizientenmatrix regulär, so wird diese transformiert auf die Einheitsmatrix. Erfolgt die Erweiterung durch die Einheitsmatrix, so kann mit dem Kommando die Inverse bestimmt werden.

Wir skizzieren den möglich Ablauf des Algorithmus an kleinen Beispielen.

Beispiel 1

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | 0 | Ausgangs-Rechteckmatrix $A = [A4, I]$ |
| 1 | 1 | 0 | 1 | |
| 1 | 1 | 1 | 0 | Pivotel. $a_{11} = 1$, andere Elemente in der Spalte $\rightarrow 0$ |
| 0 | 0 | -1 | 1 | Normierung $a_{23} \rightarrow 1$ |
| 1 | 1 | 1 | 0 | |
| 0 | 0 | 1 | -1 | Pivotel. $a_{23} = 1$, andere Elemente in der Spalte $\rightarrow 0$ |
| **1** | 1 | 0 | 1 | |
| 0 | 0 | **1** | -1 | |

Im Algorithmus werden, wenn notwendig, Zeilenvertauschungen vorgenommen, wobei eine einfache Spaltenpivotstrategie mit dem Test auf die jeweilige erste Nichtnullkomponente der Spalte durchgeführt wird. Die von links begrenzenden "Diagonalelemente" werden auf den Wert Eins normiert ("Eins-Front"). Links davon in den Zeilen stehen lauter Nullen, über den Diagonalelementen werden ebenfalls Nullen erzeugt. Eventuelle Zeilenvertauschungen sind aus der Darstellung jedoch nicht ersichtlich.

Beispiel 2

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | Ausgangs-Rechteckmatrix $A = [A3, b3]$ |
| 2 | 2 | 2 | 0 | |
| 1 | 2 | 1 | 1 | |
| 1 | 1 | 1 | 0 | Pivotel. $a_{11} = 1$, andere Elemente in der Spalte $\rightarrow 0$ |
| 0 | 0 | 0 | -2 | Zeilenvertauschung $2 \leftrightarrow 3$ |
| 0 | 1 | 0 | 0 | Normierung $a_{22} \rightarrow 1$ |
| 1 | 1 | 1 | 1 | |
| 0 | 1 | 0 | 0 | Pivotel. $a_{22} = 1$, andere Elemente in der Spalte $\rightarrow 0$ |
| 0 | 0 | 0 | -2 | |
| 1 | 0 | 1 | 1 | |
| 0 | 1 | 0 | 0 | |
| 0 | 0 | 0 | -2 | Normierung $a_{24} \rightarrow 1$ |
| 1 | 0 | 1 | 1 | |
| 0 | 1 | 0 | 0 | |
| 0 | 0 | 0 | 1 | Pivotel. $a_{24} = 1$, andere Elemente in der Spalte $\rightarrow 0$ |
| **1** | 0 | 1 | 0 | |
| 0 | **1** | 0 | 0 | |
| 0 | 0 | 0 | **1** | |

Die $LU$-Faktorisierung mit dem Kommando `lu` liefert ohne Zeilenvertauschung $A = LU$ mit den entsprechen Dreiecksmatrizen $L$, $l_{ii} = 1$, und $U$. Zeilenvertauschungen erfolgen gemäß einer Spaltenpivotstrategie mit dem Test auf die jeweilige betragsgrößte Komponente der Spalte. Sie werden entweder in die untere Dreiecksmatrix $L$ einbezogen (permutierte untere Dreiecksmatrix, psychologically lower triangular matrix) oder als Permutationsmatrix in Form eines zusätzlichen Ergebnisparameters angegeben.

| | | |
|---|---|---|
| $[L, U] = lu(A)$ | $A = LU$, | $L$ permutierte untere Dreiecksmatrix, $U$ obere Dreiecksmatrix |
| $[L, U, P] = lu(A)$ | $PA = LU$, | $L, U$ untere bzw. obere Dreiecksmatrix, $P$ Permutationsmatrix |

Beispiel 3

```
A = [2 3 1; 1 1 1; 1 2 1];
lu(A)
```

```
ans =
    2.0000    3.0000    1.0000
   -0.5000   -0.5000    0.5000
   -0.5000    1.0000    1.0000
```

Wenn die $LU$-Zerlegung wie hier ohne Zeilenvertauschung abläuft, kann man aus ihrem Ergebnistableau (wie Tableau $C\backslash B$ beim verketteten Gaußalgorithmus) die beiden Dreiecksmatrizen herauslesen.

$$L = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{2} & -1 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 2 & 3 & 1 \\ 0 & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix}$$

Beispiel 4

```
A = [1 1 1; 2 3 1; 1 2 1];
lu(A)

ans =
    2.0000    3.0000    1.0000
   -0.5000   -0.5000    0.5000
   -0.5000    1.0000    1.0000
```

Man kann vermuten, und davon muß man im allgemeinen ausgehen, daß Zeilenvertauschungen stattfinden. Somit gilt

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 3 & 1 \\ 1 & 2 & 1 \end{pmatrix} = P^T \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{2} & -1 & 1 \end{pmatrix} \begin{pmatrix} 2 & 3 & 1 \\ 0 & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix}$$

mit der Permutationsmatrix $P$. In diesem Fall ist die Anwendung der Kommandos in der Form `[L U] = lu(A)` oder `[L U P] = lu(A)` ($PA = LU$) aufschlußreicher.

```
[L U] = lu(A)

L =
    0.5000    1.0000         0
    1.0000         0         0
    0.5000   -1.0000    1.0000

U =
    2.0000    3.0000    1.0000
         0   -0.5000    0.5000
         0         0    1.0000
```

Des weiteren berücksichtigen wir die Invertierung der Matrix mittels `inv` sowie den speziellen Array-Operator \ zur Lösung von LGS.

Benutzen wir also die genannten Kommandos für weitere Beispiele.

```
% Definition der Komponenten von LGS
A1 = [1 1 1; 2 3 1; 1 2 1];
A2 = [1 2 -3 1; 0 2 3 -1; 4 2 -1 3; -7 -2 -1 -5];
A3 = [1 1 1; 2 2 2; 1 2 1];
A4 = [1 1; 1 1];
A5 = [0 1 2; 1 2 3; 2 3 5];
b1 = [1 2 3]';
b2 = [b1; 4];
b3 = [1 0 1]';
```

### Matrixrang `rank` und Gaußalgorithmus `rref` (Row reduce Variante)

```
% LGS 1
length(A1)

ans =
     3

rank(A1)

ans =
     3

rref([A1 b1])

ans =
     1     0     0    -3
     0     1     0     2
     0     0     1     2

% LGS 2, A2 ist singulaer
% keine Loesung wegen Rangunterschied  rang(A2)<rang(A2,b2)
rank(A2)

ans =
     3

rank([A2,b2])

ans =
     4

rref([A2 b2])
```

```
ans =
    1.0000         0         0    0.8000         0
         0    1.0000         0   -0.2000         0
         0         0    1.0000   -0.2000         0
         0         0         0         0    1.0000

% LGS 3, A3 ist singulaer
% Situation wie bei LGS 2
rank(A3)

ans =
     2

rank([A3 b3])

ans =
     3

rref([A3 b3])

ans =
     1     0     1     0
     0     1     0     0
     0     0     0     1

% LGS 4
% unendlich viele Loesungen wegen  rang(A3)=rang(A3,b1)
rank([A3 b1])
rref([A3 b1])

ans =
     1     0     1    -1
     0     1     0     2
     0     0     0     0
```

### Matrixinvertierung mit `inv` und Gaußalgorithmus `rref`

Zeilenvertauschungen haben bzg. des Herauslesens der inversen Matrix $A^{-1}$ aus dem Gesamttableau `rref([A eye(lenght(A))])` keine Bedeutung.

```
inv(A1)

ans =
     1     1    -2
    -1     0     1
     1    -1     1
```

```
rref([A1 eye(3)])

ans =
     1     0     0     1     1    -2
     0     1     0    -1     0     1
     0     0     1     1    -1     1

% inv(A1)=
ans(1:3,4:6);

% Invertierung von A2, A2 ist singulaer
% 4.Zeile des Tableaus zeigt auf das Problem
rref([A2 eye(4)])

ans =
  Columns 1 through 7
    1.0000         0         0    0.8000         0   -0.1000   -0.1000
         0    1.0000         0   -0.2000         0    0.2750    0.5250
         0         0    1.0000   -0.2000         0    0.1500   -0.3500
         0         0         0         0    1.0000         0   -2.0000
  Column 8
   -0.2000
    0.3000
   -0.2000
   -1.0000

% Test mit inv(A2)
% Anbieten einer numerischen Loesung mit Warnung
inv(A2)

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 5.607187e-018

ans =
  1.0e+015 *
    3.6029    0.0000   -7.2058   -3.6029
   -0.9007    0.0000    1.8014    0.9007
   -0.9007    0.0000    1.8014    0.9007
   -4.5036         0    9.0072    4.5036


% Invertierung von A4, A4 ist singulaer
% 2.Zeile des Tableaus zeigt auf das Problem
rref([A4 eye(2)])
```

```
ans =
     1     1     0     1
     0     0     1    -1

% Test mit inv(A4)
% Anbieten einer numerischen Loesung (keine Loesung)
% mit eindringlicher Warnung
inv(A4)

Warning: Matrix is singular to working precision.

ans =
   Inf   Inf
   Inf   Inf

% Invertierung von A5, A5 ist regulaer
rref(A5)  % mit Zeilenvertauschungen

ans =
     1     0     0
     0     1     0
     0     0     1

rref([A5 eye(3)])

ans =
     1     0     0    -1    -1     1
     0     1     0    -1     4    -2
     0     0     1     1    -2     1

% ans(1:3,4:6)=inv(A5)
```

### Lösung von LGS mit Array-Operator \

```
% LGS 1
x = A1\b1

x =
    -3
     2
     2

[L U] = lu(A1);   % siehe oben, A1=LU
x = U\(L\b1);     % x=U^(-1)L^(-1)b1
```

```
% Achtung, y ist keine Loesung des LGS
y = U\L\b1        %  wie  y=(U\L)\b1=L^(-1)Ub1

y =
    0.5000
   10.7500
   13.5000

% LGS 2, A2 ist singulaer
% Anbieten einer numerischen Loesung mit Warnung
x = A2\b2

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 5.607187e-018

x =
  1.0e+016 *
   -3.2426
    0.8106
    0.8106
    4.0532

% gleiche Warnung und Ergebnis mit
x = inv(A2)*b2;

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 5.607187e-018

% Versuch einer Faktorisierung, dann Inverse
% gleiche Warnung bzg. oberer Dreiecksmatrix U
[L U] = lu(A2)

L =
   -0.1429    0.8571    1.0000         0
         0    1.0000         0         0
   -0.5714    0.4286    0.5000    1.0000
    1.0000         0         0         0

U =
   -7.0000   -2.0000   -1.0000   -5.0000
         0    2.0000    3.0000   -1.0000
         0         0   -5.7143    1.1429
         0         0         0    0.0000

inv(L)   % L regulaer
```

```
ans =
         0         0         0    1.0000
         0    1.0000         0         0
    1.0000   -0.8571         0    0.1429
   -0.5000         0    1.0000    0.5000

inv(U)   % U singulaer

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 5.194894e-018

ans =
  1.0e+015 *
    0.0000    0.0000    0.0000   -7.2058
         0    0.0000    0.0000    1.8014
         0         0    0.0000    1.8014
         0         0         0    9.0072

x = inv(U)*inv(L)*b2;  % Loesung wie oben

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 5.194894e-018

x = U\(L\b2);          % Loesung wie oben

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.586033e-017

% LGS 3, A3 ist singulaer
% Keine numerische Loesung mit ernster Warnung
x = A3\b3

Warning: Matrix is singular to working precision.

x =
   Inf
   Inf
   Inf
```

## 2.2 Lösung überbestimmter Gleichungssysteme

Für die Invertierung einer regulären quadratischen Matrix $A(n,n)$ kennen wir das Kommando `inv`. Ist die Matrix jedoch nicht quadratisch oder quadratisch und singulär, muß auf die Pseudoinverse zurückgegriffen werden.
Der Begriff der Pseudoinversen oder Moore-Penrose-Inversen $A^+ \in \mathbb{R}^{n,m}$ der Rechteckmatrix $A \in \mathbb{R}^{m,n}$ ist mit einer Reihe von weiteren Aspekten der numerischen linearen Algebra verbunden:

- Quadratmittelproblem und -lösung,
- Singulärwertzerlegung (SVD),
- Normalgleichungssystem,
- Minimum-Norm-Lösung oder Normallösung.

Ich beschränke mich hier auf eine kurze Beschreibung der Situation und auf überbestimmte Gleichungssysteme mit $A \in \mathbb{R}^{m,n},\ m > n \geq k = \text{rang}(A)$.
Wenn $k = n$ ist, dann ist $A$ spaltenregulär und die Matrix $A^T A$ des Normalgleichungssystems ist symmetrisch positiv definit und damit regulär.

### Quadratmittelproblem

Dies ist ein abgeschwächtes Ersatzproblem für das LGS notiert in der Form

$$\|b - Ax\| \ \rightarrow \min_{x \in \mathbb{R}^n} \quad \text{bzw.} \quad Ax \cong b.$$

Ein Vektor $x \in \mathbb{R}^n$, für den gilt

$$\|b - Ax\| \leq \|b - Ax'\| \quad \forall x' \in \mathbb{R}^n,$$

heißt Quadratmittellösung des LGS $Ax = b$.
Das Problem ist für jede rechte Seite $b$ lösbar. Es können durchaus unendlich viele Lösungen existieren.

### Singulärwertzerlegung (SVD)

Das ist eine Zerlegung der Matrix $A \in \mathbb{R}^{m,n}$ in der Form

$$A = U\ \Sigma\ V^T$$

mit $\Sigma = \text{diag}(\sigma_1, ..., \sigma_k, 0, ..., 0) \in \mathbb{R}^{m,n},\ \sigma_1 \geq ... \geq \sigma_k > 0 = \sigma_{k+1} = ... = \sigma_n,\ k = \text{rang}(A)$, und den orthogonalen Matrizen $(H^T H = H H^T = I)\ U \in \mathbb{R}^{m,m}$ und $V \in \mathbb{R}^{n,n}$.

Jede Matrix besitzt eine SVD. Die Singulärwerte $\sigma_i,\ i = 1, ..., n$, sind die Wurzeln der nichtnegativen Eigenwerte von $A^T A$ und sind eindeutig bestimmt.
Für diese wichtige Methode der linearen Algebra bietet MATLAB das Kommando `svd`.

```
% Singular Value Decomposition
A = [1 -1; 1 0; 1 1];

[ U S V] = svd(A)
```

```
U =
    0.5774    0.7071    0.4082
    0.5774    0.0000   -0.8165
    0.5774   -0.7071    0.4082

S =
    1.7321         0
         0    1.4142
         0         0

V =
     1     0
     0    -1

% Vergleich von USV' mit A
U*S*V'

ans =
    1.0000   -1.0000
    1.0000    0.0000
    1.0000    1.0000


disp('Singulaerwerte');
diag(S)

ans =
    1.7321
    1.4142

% Vergleich mit Kommando fuer Eigenwertberechnung  eig
D = eig(A'*A)

D =
     3
     2

sqrt(D)

ans =
    1.7321
    1.4142
```

**Normalgleichungssystem**

Es hat die Form

$$A^T A x = A^T b$$

und ergibt sich aus dem Minimierungsproblem (Methode der kleinsten Fehlerquadrate)

$$(b - Ax)^T (b - Ax) \;\rightarrow\; \min_{x \in \mathbb{R}^n}.$$

Eine eindeutige Lösung finden wir im Fall $k = \text{rang}(A) = n$. Andernfalls können unendlich viele Vektoren die Norm des Residuums $r(x) = b - Ax$ minimieren. Dann ist es sinnvoll, unter diesen eine spezielle Lösung auszuzeichnen. Die Minimum-Norm-Lösung (Normallösung) ist eine solche eindeutige in der Euklidischen Norm.

**Pseudoinverse**

- Zu jeder Matrix $A \in \mathbb{R}^{m,n}$ gibt es genau eine Matrix $A^+ \in \mathbb{R}^{n,m}$, so daß $A^+ b$ die Normallösung von $Ax \cong b$ darstellt.
- Wenn $A = U\ \Sigma\ V^T$ ist, dann gilt $A^+ = V\ \Sigma^+\ U^T \in \mathbb{R}^{n,m}$,
  $\Sigma^+ = \text{diag}(\frac{1}{\sigma_1}, ..., \frac{1}{\sigma_k}, 0, ..., 0) \in \mathbb{R}^{n,m}$.
- $A^+$ ist durch die Penrose-Bedingungen

  $$AA^+A = A,\ A^+AA^+ = A^+,\ (AA^+)^T = AA^+,\ (A^+A)^T = A^+A$$

  eindeutig festgelegt.

In diesem Zusammenhang bietet MATLAB zwei Möglichkeiten der Bestimmung von Quadratmittellösungen für überbestimmte LGS. Ist die Quadratmittellösung eindeutig, führen beide zum selben Ergebnis.

- Normallösung mittels Kommando `pinv`
  $x = pinv(A) * b$
- Quadratmittellösung mittels Array-Operator \
  $y = A \backslash b$
  Dabei wird der Rang $k$ der Matrix $A$ in der zugrundeliegenden $QR$-Zerlegung mitbestimmt. Der Vektor $y$ hat dann höchstens $k$ Nichtnullkomponenten.
  In der Euklidischen Norm gelten dann
  - $\|b - Ax\| = \|b - Ay\|$
  - $\|x\| \leq \|y\|$
  - $y$ ist unter allen Quadratmittellösungen diejenige mit der geringsten Anzahl von Nichtnullkomponenten (also meisten Nullen).

```
% Loesung eines ueberbestimmten Gleichungssystems
% mit spaltensingulaerer Koeffizientenmatrix
A = magic(8);
A = A(:,1:6)    % rang(A)=3

A =
    64     2     3    61    60     6
     9    55    54    12    13    51
    17    47    46    20    21    43
    40    26    27    37    36    30
    32    34    35    29    28    38
    41    23    22    44    45    19
    49    15    14    52    53    11
     8    58    59     5     4    62

b = 260*ones(1,8)

b =
   260   260   260   260   260   260   260   260

b = b';

% Singulaerwertzerlegung
[ U S V] = svd(A)

U =
  Columns 1 through 7
    0.3554   -0.5585    0.3215    0.0171    0.4495   -0.4591    0.2126
    0.3517    0.4047   -0.3336    0.3793   -0.2339   -0.4739    0.4206
    0.3517    0.2507   -0.3421    0.1916    0.6615    0.1964   -0.4021
    0.3554   -0.0963    0.3470    0.2516   -0.4262    0.0492   -0.3845
    0.3554    0.0578    0.3554    0.3834   -0.0924    0.3875    0.0211
    0.3518   -0.2115   -0.3675   -0.3868   -0.3118   -0.2702   -0.4809
    0.3518   -0.3656   -0.3760   -0.1842   -0.1158    0.5477    0.4625
    0.3553    0.5200    0.3809   -0.6521    0.0690    0.0224    0.1508
  Column 8
    0.0000
   -0.0333
   -0.1478
   -0.5906
    0.6618
    0.3811
   -0.2000
   -0.0712
```

```
S =
  225.1696         0         0         0         0         0
         0  127.1865         0         0         0         0
         0         0   11.7579         0         0         0
         0         0         0    0.0000         0         0
         0         0         0         0    0.0000         0
         0         0         0         0         0    0.0000
         0         0         0         0         0         0
         0         0         0         0         0         0

V =
    0.4085   -0.4110    0.5583   -0.0485    0.1777    0.5645
    0.4080    0.4104   -0.3983   -0.6313    0.1229    0.3046
    0.4081    0.4092   -0.1582    0.7478    0.2462    0.1457
    0.4083   -0.4073   -0.1623   -0.0880    0.5197   -0.6027
    0.4082   -0.4061   -0.4025    0.1365   -0.6974    0.0382
    0.4083    0.4055    0.5624   -0.1165   -0.3692   -0.4502

% Vergleich mit A
% USV' stimmt mit A bis auf die ev. letzte Position der
% 16-stelligen Mantisse ueberein
U*S*V';

% Normalloesung mittels der Pseudoinversen
k = rank(S);    % =3
Sp = S';
for i=1:k Sp(i,i)=1/Sp(i,i); end;
Sp

Sp =
  Columns 1 through 7
    0.0044         0         0         0         0         0         0
         0    0.0079         0         0         0         0         0
         0         0    0.0850         0         0         0         0
         0         0         0    0.0000         0         0         0
         0         0         0         0    0.0000         0         0
         0         0         0         0         0    0.0000         0
  Column 8
         0
         0
         0
         0
         0
         0
```

```
% MOORE-PENROSE-Inverse

Ap = pinv(A)    % bzw.  Ap=U*Sp*U'

Ap =
  Columns 1 through 7
    0.0177   -0.0165   -0.0164    0.0174    0.0173   -0.0161   -0.0160
   -0.0121    0.0132    0.0130   -0.0114   -0.0112    0.0124    0.0122
   -0.0055    0.0064    0.0060   -0.0043   -0.0040    0.0049    0.0045
   -0.0020    0.0039    0.0046   -0.0038   -0.0044    0.0064    0.0070
   -0.0086    0.0108    0.0115   -0.0109   -0.0117    0.0139    0.0147
    0.0142   -0.0140   -0.0149    0.0169    0.0178   -0.0176   -0.0185

  Column 8
    0.0170
   -0.0106
   -0.0028
   -0.0063
   -0.0141
    0.0205

x = Ap*b;
x'

ans =
    1.1538    1.4615    1.3846    1.3846    1.4615    1.1538

% Residuum
norm(A*x-b)

ans =
     0

% Quadratmittelloesung
y = A\b;

Warning: Rank deficient, rank = 3  tol =   1.8829e-013

y'

ans =
    3.0000    4.0000         0         0    1.0000         0

% Residuum
norm(A*y-b)
```

```
ans =
  9.8456e-014

y1 = [3 4 0 0 1 0]';    % exakt
norm(A*y1-b)

ans =
     0

% Normalgleichungssystem loesen
% rank(A'*A)=rank([A'*A A'*b])=3

z = (A'*A)\(A'*b);

Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 3.588452e-019

z'

ans =
   -0.3153  -12.6938   -4.0202  -65.5743   69.8897   20.7139

% Residuum
norm(A*z-b)

ans =
  3.0079e-013

% Vergleich der Euklidischen Normen

nx = num2str(norm(x));    % 4 digits of precision
ny = num2str(norm(y));
nz = num2str(norm(z));
disp(['|x|=',nx,'<=|y|=',ny,'<=|z|=',nz]);

|x|=3.282<=|y|=5.099<=|z|=98.95
```

# 3 Matrixeigenwertproblem

## 3.1 Grundlagen und Verfahren

Für die reelle oder komplexe $(n,n)$-Matrix $A$ heißt $\lambda \in \mathbb{C}$ *Eigenwert* (EW) von $A$, falls ein Vektor $x \in \mathbb{C}^n,\ x \neq 0$, existiert, so daß $Ax = \lambda x$ gilt. $x$ heißt *Eigenvektor* (EV).

- $\lambda$ ist Eigenwert von $A$ gdw. $\lambda$ ist Nullstelle des *charakteristischen Polynoms*

$$p(\lambda) = det(A - \lambda I) = p_n\lambda^n + p_{n-1}\lambda^{n-1} + p_{n-2}\lambda^{n-2} + ... + p_1\lambda + p_0,\ p_n = (-1)^n.$$

- Die Menge der $n$ Eigenwerte $\sigma(A) = \{\lambda_1, \lambda_2, ..., \lambda_n\}$ heißt *Spektrum* von $A$. Der *Spektralradius* ist $\rho = \max\limits_i\ |\lambda_i|$ .

- $\lambda_i$ besitzt die *algebraische Vielfachheit* $n_i \geq 1$, falls $\lambda_i$ eine $n_i$-fache Nullstelle von $p(\lambda)$ ist.

- $\lambda_i$ besitzt die *geometrische Vielfachheit* $m_i \geq 1$ , falls $m_i = n - Rang(A - \lambda I)$ gilt. Zu $\lambda_i$ existieren dann genau $m_i$ linear unabhängige Eigenvektoren. Es ist stets $m_i \leq n_i$.

- Die Eigenvektoren sind bis auf einen Proportionalitätsfaktor festgelegt.

Beim Eigenwertproblem geht es um die Bestimmung aller $n$ Eigenwerte $\lambda_1, \lambda_2, ..., \lambda_n$ und evtl. aller Eigenvektoren (vollständiges EWP) bzw. eines oder weniger ausgewählter Eigenwerte (unvollständiges, partielles EWP).
Das EWP ist besonders einfach zu handbaben für reelle Matrizen, die diagonale, tridiagonale Struktur oder Dreiecksform haben.
Das charakterisches Polynom einer Tridiagonalmatrix ist

$$p(\lambda) = det \begin{pmatrix} a_1-\lambda & b_1 & 0 & \dots & & & 0 \\ c_1 & a_2-\lambda & b_2 & 0 & \dots & & \\ 0 & c_2 & a_3-\lambda & b_3 & 0 & \dots & \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ & & & \dots & 0 & c_{n-2} & a_{n-1}-\lambda & b_{n-1} \\ 0 & & & & \dots & 0 & c_{n-1} & a_n-\lambda \end{pmatrix},$$

seine Notation in der Normalform

$$F_n = p(\lambda) = (-1)^n(q_n\lambda^n + q_{n-1}\lambda^{n-1} + ... + q_1\lambda + q_0),\quad q_n = 1.$$

Die Berechnung des Funktionswertes im Zusammenhang mit der Nullstellenaufgabe $p(\lambda) = 0$ erfolgt über die Entwicklung der Determinante nach der letzten Spalte und dann nach der letzten Zeile. Das führt auf die rekursive Formel (3-Term-Rekursion)

$$F_k = (a_k - \lambda)F_{k-1} - b_{k-1}c_{k-1}F_{k-2},\ k = 2, 3, ..., n,\ F_0 = 1,\ F_1 = a_1 - \lambda.$$

Die Lösung des EWP mittels direkter Verfahren über das charakteristische Polynom ist für kleine Matrixdimensionen oder spezielle Matrizen, z.B. mit Tridiagonalstruktur, möglich.

Empfehlenswert sind zumeist iterative Verfahren, wie Vektoriteration, Jacobi-Verfahren mit Rotationsmatrizen oder $QR$-Transformation. Dabei ist das Ziel, eine Folge von transformierten Matrizen zu erzeugen, wo die EW sich nicht verändern und die "Grenzmatrix" von einfacher Struktur ist. Dies kann erfolgen mittels Ähnlichkeitstransformation der Gestalt $T^{-1}AT$ oder der speziellen mit einer orthogonalen Transformationsmatrix $T$ ($T^TT = TT^T = I$). Eine wichtige Rolle spielen Rotationsmatrizen (beispielhaft $n = 8$)

$$R = \begin{pmatrix} 1 & & & & & & & \mathbf{0} \\ & 1 & & & & & & \\ & & \cos\varphi & & \sin\varphi & & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ & & -\sin\varphi & & \cos\varphi & & & \\ & & & & & & 1 & \\ \mathbf{0} & & & & & & & 1 \end{pmatrix} \begin{matrix} \\ \\ \rightarrow\ p\text{-te Zeile} \\ \\ \\ \rightarrow\ q\text{-te Zeile} \\ \\ \\ \end{matrix} \qquad R^TR = I,$$

die beim Jacobi-Verfahren verwendet werden.

Für reelle symmetrische Matrizen, die nur reelle EW haben, deren EV ein Orthogonalsystem bilden und die damit durch eine orthogonale Ähnlichkeitstransformation in Diagonalgestalt überführbar sind, sei der Algorithmus des klassischen Jacobi-Verfahrens noch kurz erläutert.

## 3.2 Das Jacobi-Verfahren

Ziel ist die Überführung von $A = A^T$ in eine Diagonalmatrix $D = diag(\lambda_1, \lambda_2, ..., \lambda_n)$ durch orthogonale Ähnlichkeitstransformation $D = T^TAT$, $T = R_1R_2...R_k...$, $T^TT = I$, und Rotationsmatrizen $R_k$. Die Spalten von $T$ sind die orthogonalen EV von $A$.
Aufgabe jedes Rotationsschritts ist es, ein Außendiagonalelement $a_{pq} \neq 0$ (Pivotelement) zu Null zu machen. Es gilt

$$A_0 = A,\ A_k = R_k^TA_{k-1}R_k, \quad A_k = T_k^TAT_k \quad \text{mit} \quad T_k = R_1R_2...R_k,$$
$$T_0 = I, \quad T_k = T_{k-1}R_k, \quad k = 1, 2, 3, ...$$

und

$$A_k = (a_{ij}^{(k)}), \quad S(A_k) = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} (a_{ij}^{(k)})^2, \quad \lim_{k\to\infty} S(A_k) = 0.$$

**Berechnung der Werte** $\cos\varphi$ **und** $\sin\varphi$

$$\theta = \frac{a_{qq} - a_{pp}}{2a_{pq}}$$

$$\tan\varphi = t = \begin{cases} \dfrac{1}{\theta + \text{sign}(\theta)\sqrt{\theta^2+1}} & \text{für} \quad \theta \neq 0 \\ 1 & \text{für} \quad \theta = 0 \end{cases}$$

$$\cos\varphi = \frac{1}{\sqrt{1+t^2}}, \qquad \sin\varphi = \frac{t}{\sqrt{1+t^2}}$$

**Transformationsformeln**

**Zeilen $p$ und $q$**

$$\begin{aligned} a'_{pj} &= a_{pj}\cos\varphi - a_{qj}\sin\varphi \\ a'_{qj} &= a_{pj}\sin\varphi + a_{qj}\cos\varphi, \quad j = 1(1)n,\ j \neq p,\ j \neq q \end{aligned}$$

**Spalten $p$ und $q$**

$$\begin{aligned} a'_{ip} &= a_{ip}\cos\varphi - a_{iq}\sin\varphi \\ a'_{iq} &= a_{ip}\sin\varphi + a_{iq}\cos\varphi, \quad i = 1(1)n,\ i \neq p,\ i \neq q \end{aligned}$$

**4 Kreuzungspunkte**

$$\begin{aligned} a'_{pp} &= a_{pp}\cos^2\varphi - 2a_{pq}\cos\varphi\sin\varphi + a_{qq}\sin^2\varphi \\ a'_{qq} &= a_{pp}\sin^2\varphi + 2a_{pq}\cos\varphi\sin\varphi + a_{qq}\cos^2\varphi \\ a'_{pq} &= a'_{qp} = (a_{pp} - a_{qq})\cos\varphi\sin\varphi + a_{pq}(\cos^2\varphi - \sin^2\varphi) = 0 \end{aligned}$$

**Auswahlregeln für Pivotelement**

- Klassisches Verfahren
  $a_{pq}$ ist das betragsgrößte Außendiagonalelement von $A$.
- Zyklisches Verfahren
  Durchlauf der Pivotelemente oberhalb der Hauptdiagonalen in einer vorgegebenen Reihenfolge, z.B.
  $a_{12}, a_{13}, ..., a_{1n}, a_{23}, ..., a_{n-1,n},\ a_{12}, a_{13}, ...$, keine Rotation, wenn $a_{pq} = 0$.

## 3.3 EWP in MATLAB

Zuerst einige symbolische Berechnungen mit der Rotationsmatrix $R(2,2)$, die ohne weiteres auf den allgemeinen Fall übertragbar sind.

```
R = sym('[ cos(t) sin(t); -sin(t) cos(t) ]')

R =
[  cos(t), sin(t)]
[ -sin(t),cos(t) ]

transpose(R)            % Transposition der Matrix

ans =
[cos(t), -sin(t)]
[sin(t),  cos(t)]

symmul(R,transpose(R))  % Kontrolle R*R^T=I
```

```
ans =
[cos(t)^2+sin(t)^2,                 0]
[                0, cos(t)^2+sin(t)^2]

simplify(ans)

ans =
[1, 0]
[0, 1]
```

Die Vertauschung der $p$-ten mit der $q$-ten Zeile sowie der $p$-ten mit der $q$-ten Spalte in der Matrix $A$ kann mit einer Permutationsmatrix $P$, $PP^T = I$, gemäß $B = PAP^T = PAP^{-1}$ erfolgen. Die EW sind invariant bei dieser Ähnlichkeitstransformation. Wegen

$$By = \lambda y,\ B = PAP^T,\ \ y = Px,$$

$$Ax = (P^TP)A(P^TP)x = P^T(PAP^T)Px = P^TBy = \lambda P^Ty = \lambda x$$

wird in den EV von $B$ nur die Reihenfolge der Komponenten vertauscht.

```
n = 4;
A = rand(n)

A =
    0.4523    0.2152    0.8609    0.8176
    0.8089    0.6796    0.4713    0.7558
    0.9317    0.9089    0.5060    0.4622
    0.6516    0.2501    0.6004    0.9514

P = diag(ones(n,1));
p = 1;  q = 3;
P(p,p) = 0; P(q,q) = 0; P(p,q) = 1; P(q,p) = 1;
P

P =
     0     0     1     0
     0     1     0     0
     1     0     0     0
     0     0     0     1

B = P*A*P'

B =
    0.5060    0.9089    0.9317    0.4622
    0.4713    0.6796    0.8089    0.7558
    0.8609    0.2152    0.4523    0.8176
    0.6004    0.2501    0.6516    0.9514
```

```
eig(A)     % =eig(B)

ans =
   2.5602
  -0.2788
   0.1539 + 0.1494i
   0.1539 - 0.1494i
```

Um sich das Jacobi-Verfahren verständlich zu machen, genügt es, eine Rotation auszuführen. Für akademische Beispiele kann bei exakter Rechnung das Verfahren durchaus nach endlich vielen Schritten beendet sein. Bei seiner Realisierung in einer Gleitkommaarithmetik ist dies eher unwahrscheinlich.

```
% Ein Schritt des Jacobi-Verfahrens
M3 = [ 2 -1  0
      -1  2 -1
       0 -1  2 ];

disp('1 Rotation:  M3(1,2)=M3(2,1) -> 0')
p = 1; q = 2;
theta = (M3(q,q)-M3(p,p))/(2*M3(p,q))

theta =
     0

if theta==0
  t = 1
    elseif theta>0
  t = 1/(theta+sqrt(theta^2+1))
  else
  t = 1/(theta-sqrt(theta^2+1))
end

t =
     1

co = 1/sqrt(1+t^2)

co =
    0.7071

si = t/sqrt(1+t^2)

si =
    0.7071
```

```
R = [  co si  0
      -si co  0
        0  0  1 ];

B = R'*M3*R

B =
    3.0000         0    0.7071
         0    1.0000   -0.7071
    0.7071   -0.7071    2.0000

format long
disp('MATLAB-Funktion  eig')
EW = eig(M3)

EW =
   0.58578643762691
   2.00000000000000
   3.41421356237309

rho = max(abs(EW))

rho =
   3.41421356237309

format short
disp('exakte EW:   2-2*cos(k*pi/4), k=1,2,3')
k=1:3;
lambda = 2-2*cos(k*0.25*pi);
disp(['           ',sprintf('%10.6f',lambda)])

             0.585786  2.000000  3.414214
```

Die obigen Programmzeilen zum Jacobi-Verfahren können natürlich die Grundlage für eine entsprechende MATLAB-Funktion sein. Im $m$-File *eijacobi.m* ist das zyklisches Jacobi-Verfahren für eine reelle symmetrische $(n,n)$-Matrix implementiert. Die Pivotelemente sind $a_{ij}$, $i > j$. Aufgrund der Symmetrie der Matrix bezieht sich die Kontrollsumme nur auf die Matrixelemente unterhalb der Hauptdiagonalen.

$$s = \sum_{i=2}^{n} \sum_{j=1}^{i-1} a_{ij}^2 = \sum_{j=1}^{n-1} \sum_{i=j+1}^{n} a_{ij}^2.$$

```
% EIJACOBI.M
% Zyklisches Jacobi-Verfahren fuer EWP einer symmetrischen Matrix

function [lambda,v,s] = eijacobi(n,a,epsi)
```

```
% Eingangsparameter
%  n        Dimension der Matrix
%  a        symmetrische Matrix
%  epsi     Toleranz fuer Abbruch, Summe der Quadrate der
%           Aussendiagonalelemente unterhalb der Diagonalen
% Ergebnisse
%  lambda   Vektor der EW (i.a. Naeherungen)
%  v        Matrix mit Spalten=EV
%  s        letzte Kontrollsumme

   epsi2 = epsi^2;
   v = diag(ones(n,1));
   s = sum(sum(tril(a,-1).^2));

   while (2.0*s >= epsi2)
     for p=1:n-1
       for q=p+1:n
         if abs(a(q,p)) >= epsi2
           theta = (a(q,q)-a(p,p))/(2.0*a(q,p));
           if 1.0+theta == 1.0
             t = 1.0;
            elseif theta<0
             t = 1.0/(theta-sqrt(theta^2+1.0));
            else
             t = 1.0/(theta+sqrt(theta^2+1.0));
           end
           c = 1.0/sqrt(1.0+t^2);
           s = c*t;
           r = s/(1.0+c);

           a(p,p) = a(p,p)-t*a(q,p);
           a(q,q) = a(q,q)+t*a(q,p); a(q,p) = 0.0;

           g = a(q,1:p-1)+r.*a(p,1:p-1);
           h = a(p,1:p-1)-r.*a(q,1:p-1);
           a(p,1:p-1) = a(p,1:p-1)-s.*g;
           a(q,1:p-1) = a(q,1:p-1)+s.*h;

           g = a(q,p+1:q-1)'+r.*a(p+1:q-1,p);
           h = a(p+1:q-1,p)'-r.*a(q,p+1:q-1);
           a(p+1:q-1,p) = a(p+1:q-1,p)-s.*g;
           a(q,p+1:q-1) = a(q,p+1:q-1)+s.*h;

           g = a(q+1:n,q)+r.*a(q+1:n,p);
           h = a(q+1:n,p)-r.*a(q+1:n,q);
```

```
            a(q+1:n,p) = a(q+1:n,p)-s.*g;
            a(q+1:n,q) = a(q+1:n,q)+s.*h;

            % EV
            g = v(1:n,q)+r.*v(1:n,p);
            h = v(1:n,p)-r.*v(1:n,q);
            v(1:n,p) = v(1:n,p)-s.*g;
            v(1:n,q) = v(1:n,q)+s.*h;
          end;
        end;
      end;
      s = sum(sum(tril(a,-1).^2));

   end;
   lambda = diag(a);
% Ende Funktion eijacobi
```

**Anwendung von *eijacobi* und zum Vergleich die MATLAB-Funktion `eig`**

```
  % Zyklisches Jacobi-Verfahren

  % a) m-File eijacobi.m
  n = 4;
  A = [ 20  -7   3  -2
        -7   5   1   4
         3   1   3   1
        -2   4   1   2 ];

  epsi = 1e-2;
  [lambda,v,s] = eijacobi(n,A,epsi)

  lambda =
     23.5274
     -1.1609
      6.4605
      1.1730

  v =
      0.9106    0.1729    0.2607    0.2699
     -0.3703    0.6750    0.5876    0.2492
      0.1078   -0.1168    0.5499   -0.8200
     -0.1484   -0.7077    0.5333    0.4390

  s =
    1.3859e-009
```

```
% b) MATLAB-Funktion eig
% EW und EV ev. in anderer Reihenfolge
% EV bis auf Vorzeichen uebereinstimmend

[V,D] = eig(A)    % A*V=V*D

V =
   -0.2699   -0.1729    0.2607   -0.9106
   -0.2492   -0.6750    0.5876    0.3703
    0.8200    0.1168    0.5499   -0.1078
   -0.4390    0.7077    0.5333    0.1484

D =
    1.1730         0         0         0
         0   -1.1609         0         0
         0         0    6.4605         0
         0         0         0   23.5274
```

Schätzen wir nun die Komplexität der Funktion `eig` ab.
Sie hat die Größenordnung $\mathcal{O}(n^3)$. Zur genaueren Abschätzung kontrollieren wir am Rechner, wie die Anzahl der Operationen (`flops`) und/oder Rechenzeit (`clock, etime`) sich verhält, wenn die Matrixdimension wächst. Am besten nimmt man jeweils Verdopplungen von $n$ vor.
Die $n$-dimensionalen Testmatrizen sind

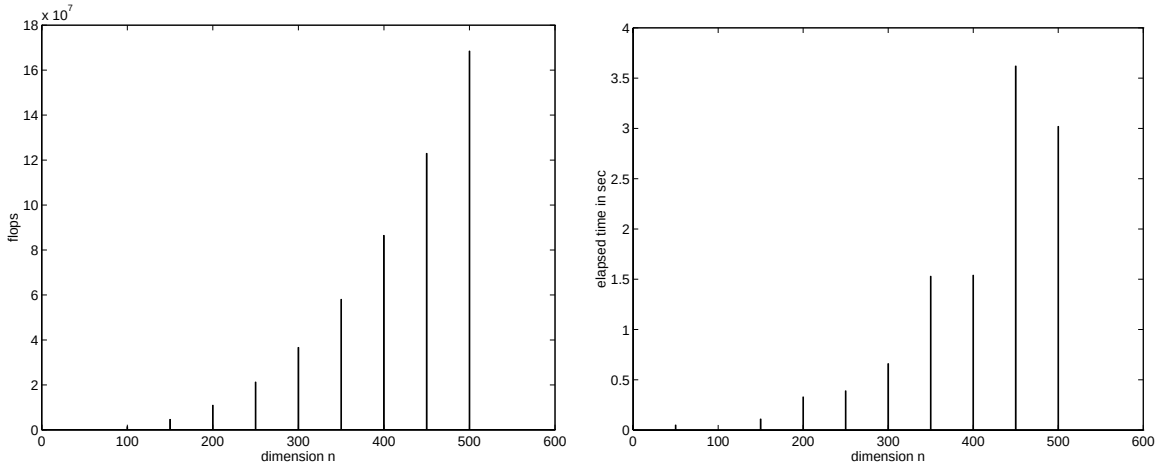
$$A = \begin{pmatrix} 2 & 3 & 4 & 5 & \dots \\ 3 & 4 & 5 & 6 & \dots \\ 4 & 5 & 6 & 7 & \dots \\ 5 & 6 & 7 & 8 & \dots \\ . & . & . & . & \dots \end{pmatrix}, \; rang(A) = 2, \qquad B = \begin{pmatrix} 0 & 1 & 1 & 1 & \dots \\ 1 & 0 & 1 & 1 & \dots \\ 1 & 1 & 0 & 1 & \dots \\ 1 & 1 & 1 & 0 & \dots \\ . & . & . & . & \dots \end{pmatrix}, \; rang(B) = n.$$

```
disp('Test der MATLAB-Funktion  eig')
n = 500;
B = ones(n)-diag(ones(n,1));
x = [2:n+1];
A = [];
for k=1:n
  A = [ A; x ];
  x = x+1;
end
A;
disp('Rangbestimmung (Geduld!)')
Rang_A = rank(A)

Rang_A =
     2
```

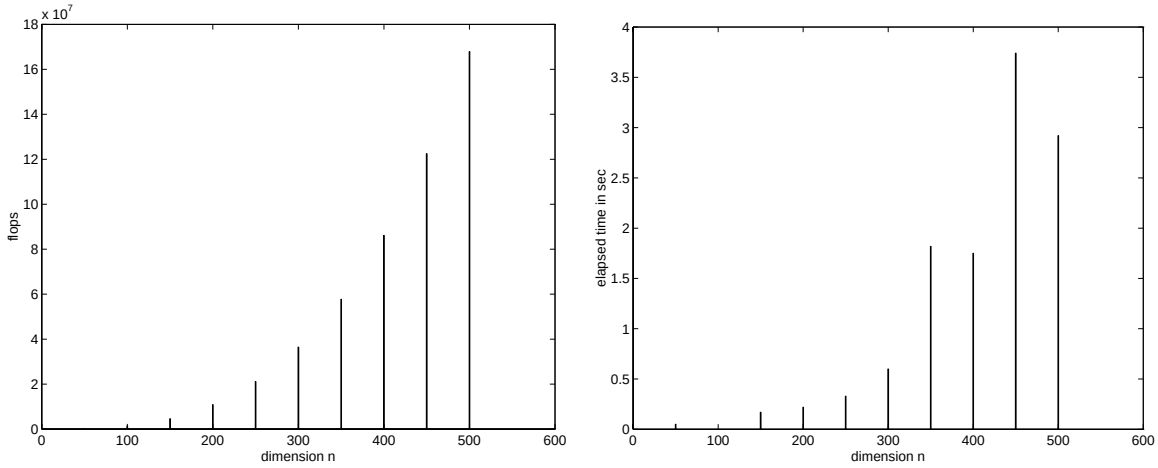
```
Rang_B = rank(B)

Rang_B =
   500

disp('Komplexitaet')
var = 1;     % und 2
for k=50:50:n
  if var == 1
     Ak = A(1:k,1:k);
   else
     Ak = B(1:k,1:k);
  end
  t0 = clock;
  flops(0);
  d = eig(Ak);
  t(k) = etime(clock,t0);
  s(k) = flops;
end
[ns,sn] = bar(s);
plot(ns,sn,'g')
xlabel('dimension n');  ylabel('flops');
if var == 1
  print ser4gr1a.ps -dps
 else
  print ser4gr1b.ps -dps
end
[nt,tn] = bar(t);
plot(nt,tn)
xlabel('dimension n');  ylabel('elapsed time in sec');
if var == 1
  print ser4gr2a.ps -dps
 else
  print ser4gr2b.ps -dps
end

% b) Schaetzen der Komplexitaet
disp('Bei Verdopplung von n waechst die Zeit')
disp('n=500 -> ca 4 sec auf PC 350MHz')
disp('100->200, 200->400, 250->500')
sprintf('%5.1f',t(200)/t(100),t(400)/t(200),t(500)/t(250))

Warning: Divide by zero
ans =
  Inf  6.2  5.2
```

Rechnungen auf PC 350MHz
Gleitkommaoperationen $flops$ und Rechenzeit in $sec$

Matrizen $A(n,n),\ n = 50(50)500$




Matrizen $B(n,n),\ n = 50(50)500$




**Zum Abschluß noch einige Bemerkungen zum $QR$-Verfahren.**

Es basiert auf der $QR$-Zerlegung der Matrix $A$ in eine orthogonale Matrix $Q$ und obere (rechte) Dreiecksmatrix $R$. Ein Schritt des Verfahrens beruht darauf, daß nach einer solchen Zerlegung das Matrixprodukt $RQ$ gebildet wird und dieses Ergebnis wieder Ausgangspunkt für den nächsten Schritt ist. Jede Matrix $B = R_m Q_m = Q_{m+1} R_{m+1}$ im Zwischenschritt ist ähnlich zu $A$, denn es gilt

$$A = Q_0 R_0, \quad R_k Q_k = Q_{k+1} R_{k+1},\ k = 0, 1, 2, ...$$

$$A = Q_0 Q_1 ... Q_m (R_m Q_m) Q_m^T ... Q_1^T Q_0^T = T B T^T, T^T T = I.$$

Die ständige Wiederholung führt letztendlich auf eine Diagonalform bzw. eine Blockdiagonalform mit $1 \times 1$- oder $2 \times 2$-Blöcken. Daraus lassen sich die reellen EW direkt sowie die komplexen EW über die Lösung einer quadratischen Gleichung mit Koeffizienten, die in den $2 \times 2$-Blöcken enthalten sind, ableiten.

Unterstützt wird der Algorithmus durch die MATLAB-Funktion `qr`.

```
% qrbasis.m
% QR-Algorithmus
function A = qrbasis(n,A,k)

% Eingangsparameter
%  n        Dimension der Matrix
%  A        Matrix
%  k        Schrittzahl des QR-Algorithmus
%
% Ergebnisse
%  A        transformierte aehnliche Matrix

for l=1:k
  [Q,R] = qr(A);
  A = R*Q;
end;
% Ende Funktion qrbasis
```

**Anwendung der Funktion** *qrbasis*

```
n = 4;
A = [ 20  -7   3  -2
      -7   5   1   4
       3   1   3   1
      -2   4   1   2 ];

disp('MATLAB-Funktion qr fuer QR-Zerlegung')
[Q,R] = qr(A)

Q =
   -0.9305   -0.1734    0.0943   -0.3086
    0.3257   -0.4975   -0.2263   -0.7715
   -0.1396   -0.4747   -0.7354    0.4629
    0.0930   -0.7050    0.6317    0.3086

R =
  -21.4942    8.3744   -2.7915    3.2102
         0   -4.5684   -3.1470   -3.5279
         0         0   -1.5180   -0.5657
         0         0         0   -1.3887

disp('Test der MATLAB-Funktion  qrbasis fuer QR-Algorithmus')
k = 20;  % auch fuer groessere k keine Diagonalform von A
A = qrbasis(n,A,k);
```

```
disp('Transformierte Matrix A');
A

A =
   23.5274    0.0000    0.0000    0.0000
    0.0000    6.4605    0.0000    0.0000
    0.0000    0.0000    0.7001   -0.9382
    0.0000    0.0000   -0.9382   -0.6880

la1 = A(1,1)

la1 =
   23.5274

la2 = A(2,2)

la2 =
    6.4605

la = roots([ 1 -A(3,3)-A(4,4) A(3,3)*A(4,4)-A(4,3)*A(3,4) ]);
la3 = la(1)

la3 =
    1.1730

la4 = la(2)

la4 =
   -1.1609
```

Bei einem $2 \times 2$-Block wie in der obigen transformierten Matrix $A$ ergibt sich das "lokale" EWP

$$det \begin{pmatrix} a_{33} - \lambda & a_{34} \\ a_{43} & a_{44} - \lambda \end{pmatrix} = 0.$$

Das charakteristische Polynom ist $p(\lambda) = \lambda^2 + (-a_{33} - a_{44})\lambda + a_{33}a_{44} - a_{34}a_{43}$, dessen beide (komplexe) Nullstellen mit dem MATLAB-Kommando `roots` bestimmt werden. Man bemerke, daß trotz Diagonalähnlichkeit der Ausgangsmatrix - sie ist symmetrisch - die transformierte nicht auf Diagonalform überführt werden kann, egal wieviel Schritte $k$ man macht.

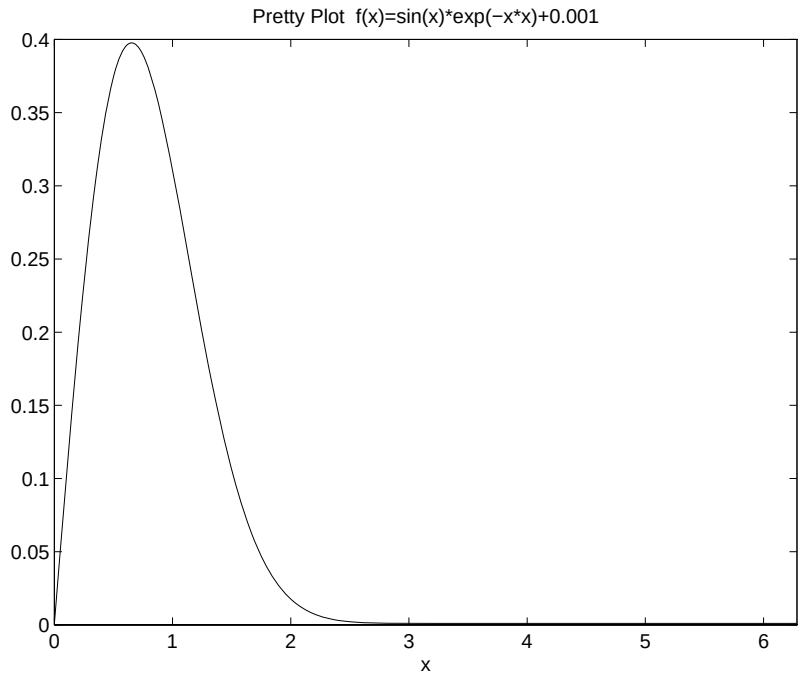
# 4 Graphische Darstellungen

Die ganzen Breite und Vielfalt der graphischen Komponenten von MATLAB soll hier nicht wiederholt werden. Es werden einige ausgewählte Plot-Befehle implementiert. Sie können dazu dienen, sich über Funktionen in der Ebene oder im Raum gewisse Vorstellungen zu verschaffen und solche Eigenschaften wie z.B. Nullstellen, Extrema, Wendepunkte, Polstellen, Höhenlinien zu untersuchen, was in den weiteren Kapiteln genutzt wird.
Wir verwenden die Plot-Kommandos `fplot, plot, fill, surf, surfl, meshgrid`. Sie besitzen zahlreiche Parameter und Optionen, von denen wir auch einige notieren.
Des weiteren sind im MATLAB-Skript Bemerkungen und Hinweise enthalten. Die Figuren werden als File (*ps*- bzw. *eps*-Format) abgelegt für die Einbeziehung in andere Dokumente.

## 4.1 Einfache 2D-Plots

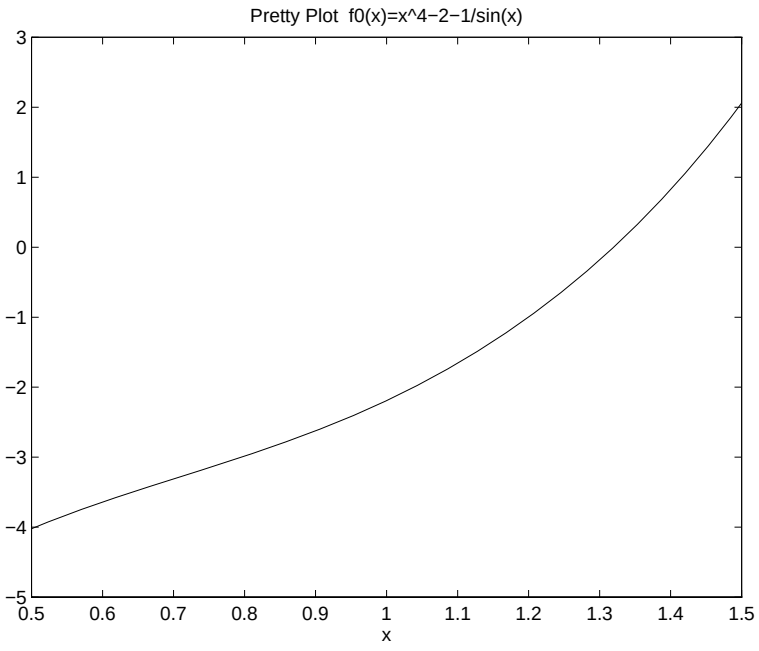
Im Zusammenhang mit graphischen Ausgaben sind in den meisten Fällen entsprechende Vektoren bereitzustellen. Somit sind solch einfache Varianten wie `plot(sin(t))` oder `plot(t,sin(t))` nicht machbar, weil eben die Argumente des Plot-Kommandos Felder sein müssen.
Trotzdem gibt es eine eher bescheidene Darstellung einer Funktion $f(x),\ x \in [a,b]$, mit `fplot`, was man als "pretty", "fast" oder "function" Plot deuten kann.

```
% Pretty/Fast Plot
f = 'sin(x)*exp(-x*x)+0.001'; % Funktionsausdruck ist Zeichenkette,
                              % Argument/unabh. Variable muss x sein
fplot(f,[0,2*pi])
title('Pretty Plot  f(x)=sin(x)*exp(-x*x)+0.001')
xlabel('x')
print graph101.ps -dps
```



Bei Ausgaben auf dem Bildschirm im Zusammenhang mit symbolischen Rechnungen wird das Bildschirmecho automatisch ausgeschaltet, deshalb ist das Kommando `echo on` des öfteren zu wiederholen.
Die Auswertung von als String gegebenen Funktionen erfolgt mit dem Substitute-Befehl `subs`, eventuell noch ergänzt durch die numerische Berechnung mit `numeric`.

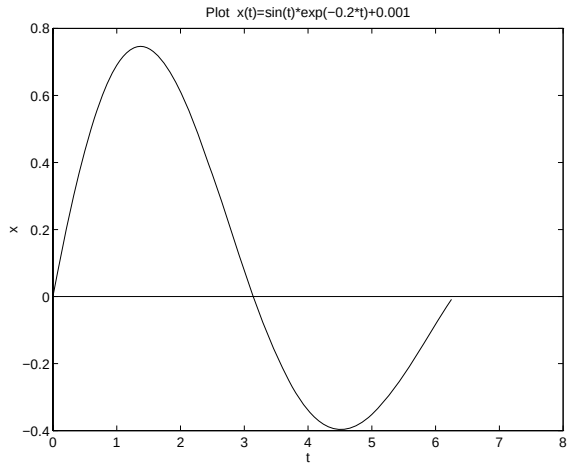
```
f0 = 'x^4-2-1/sin(x)';
f0(1)                       %  -> x als 1.Zeichen der ZK 'x^4...'

ans =
x

echo on
subs(f0,1,'x')              %  moeglich auch  subs(f0,'1','x')

ans =
-1-1/sin(1)

echo on
numeric(subs(f0,'1','x'))  %  numerische Auswertung

ans =
   -2.1884

fplot(f0,[0.5,1.5])
title('Pretty Plot  f0(x)=x^4-2-1/sin(x)')
xlabel('x')
print graph102.eps -deps
```

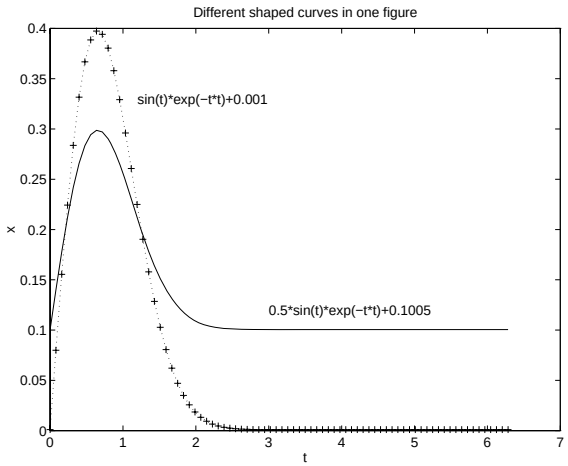


Die Grundlage für den Plot-Befehl ist die Definition von zwei gleichlangen Vektoren für die Abszissen- und Ordinatenwerte.
Dazu kann man noch Darstellungensbereiche, Achsen, Gitter, Augpunkte, Titel, Beschriftungen, Linienstile, Symbole, Farben und andere Einstellungen hinzufügen. Der Linientyp ist z.B. eine Zeichenkette der Länge $\leq 3$ mit folgenden möglichen Zeichen.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| . | point | - | solid line | y | yellow | g | green |
| o | circle | : | dotted line | m | magenta | b | blue |
| × | ×-mark | -. | dashdot line | c | cyan | w | white |
| + | plus | -- | dashed line | r | red | k | black |
| ⋆ | star | | | | | | |

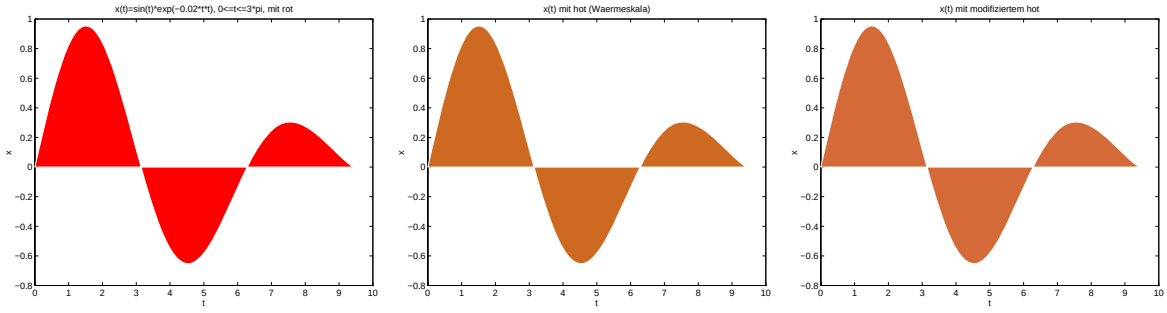
```
% Definition von 2 Vektoren: Basis fuer plot
t = 0:0.05:2*pi;
x = sin(t).*exp(-0.2.*t)+0.001;
ta = 0:0.05:8;
plot(t,x,ta,0.*ta,'w')    % mit t-Achse
title('Plot  x(t)=sin(t)*exp(-0.2*t)+0.001')
xlabel('t')
ylabel('x')
print graph103.ps -dps

% Mehrere Kurven in einer Abbildung durch Parameterliste
t = linspace(0,2*pi,80);
x1 = sin(t).*exp(-t.*t)+0.001;
x2 = 0.5.*x1+0.1;
plot(t,x1,'y:',t,x1,'+',t,x2,'g-')
title('Different shaped curves in one figure')
text(1.2,0.33,'sin(t)*exp(-t*t)+0.001')
text(3.0,0.12,'0.5*sin(t)*exp(-t*t)+0.1005')
xlabel('t')
ylabel('x')
print graph104.ps -dps
```



Füllen von 2-dimensionalen Polygonen, definiert durch die Punktfolge $(x, y)$, mit einer Farbe $c$ : `fill(x,y,c)`.

```
% Filled Polygons with different colors,
% if necessary, the polygon is closed (last=first vertex)
t = linspace(0,3*pi,100);
x = sin(t).*exp(-0.02.*t.*t);
fill(t,x,'r')   % red
title('x(t)=sin(t).*exp(-0.02.*t.*t), 0<=t<=3*pi, mit rot')
xlabel('t')
ylabel('x')
print graph105.ps -dps

colormap(hot)
fill(t,x,x)
title('x(t) mit hot (Waermeskala)')
xlabel('t')
ylabel('x')
print graph106.ps -dps

fill(t,x,x-t)
title('x(t) mit modifiziertem hot')
xlabel('t')
ylabel('x')
print graph107.ps -dps
```




## 4.2 2D-Plots auf der Basis von Funktionen in $m$-Files

Darzustellende Funktionen definiert man oft in $m$-Files.
Dabei sollte man beachten, ob eventuell vektorwertige Argumente verwendet werden sollen, die auch bei graphischen Auswertungen von Nutzen sind.
Die Betrachtung der Funktionen in den Files $ff1.m$ und $ff2.m$ macht den Unterschied deutlich.

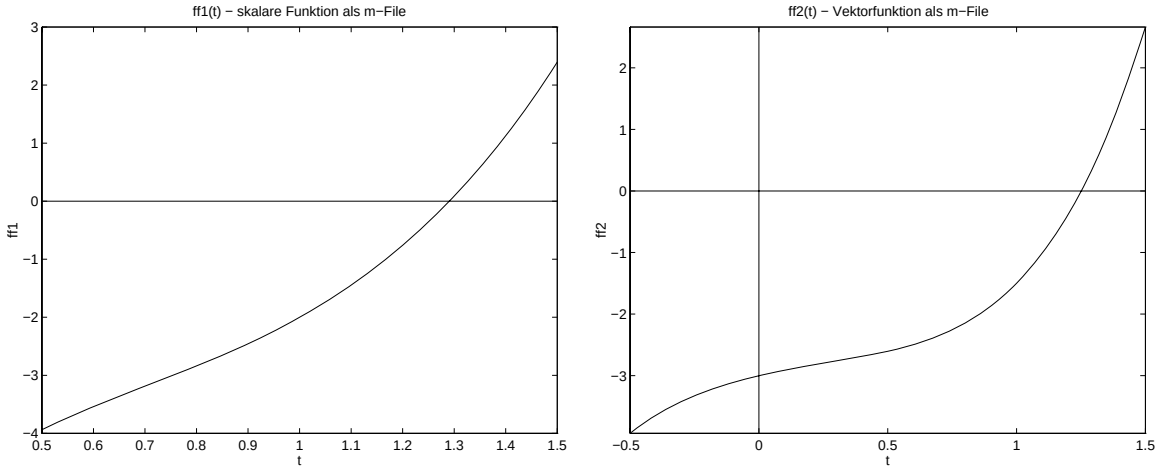
```
% ff1.m
% Skalare Funktion
```

```
function y = ff1(x)
  y = x^4-2-1/x;

% ff2.m
% Funktion mit vektorwertigem Argument
function y = ff2(x)
  y = x.^4-2-1./(x+1);
```

**Nun wenden wir beide Funktionen beim Plot an.**

```
% Arbeit mit Funktionen als m-Files
% Skalare Funktion ff1
ff1(1)

ans =
    -2

% ff1(-0.5:0.5:2.5) --> Error, matrix must be square
%  Definition der Vektoren
t = 0.5:0.05:1.5;
for i=1:max(size(t))
  x(i)= ff1(t(i));  % nicht ff1(t), da skalare Funktion
end;
plot(t,x,'y',t,0*x,'w')
title('ff1(t) - skalare Funktion als m-File')
xlabel('t')
ylabel('ff1')
print graph201.ps -dps

% Vektorfunktion ff2
ff2(1)

ans =
   -1.5000

ff2(-0.5:0.5:2.5)

ans =
   -3.9375   -3.0000   -2.6042   -1.5000    2.6625   13.6667   36.7768

%  Definition der Vektoren
t = -0.5:0.05:1.5;
x = ff2(t);
plot(t,x,'y',t,0*x,'w',[0,0],[min(x),max(x)],'w')
axis([t(1),t(max(size(t))),min(x),max(x)])
title('ff2(t) - Vektorfunktion als m-File')
```

```
xlabel('t')
ylabel('ff2')
print graph202.ps -dps
```



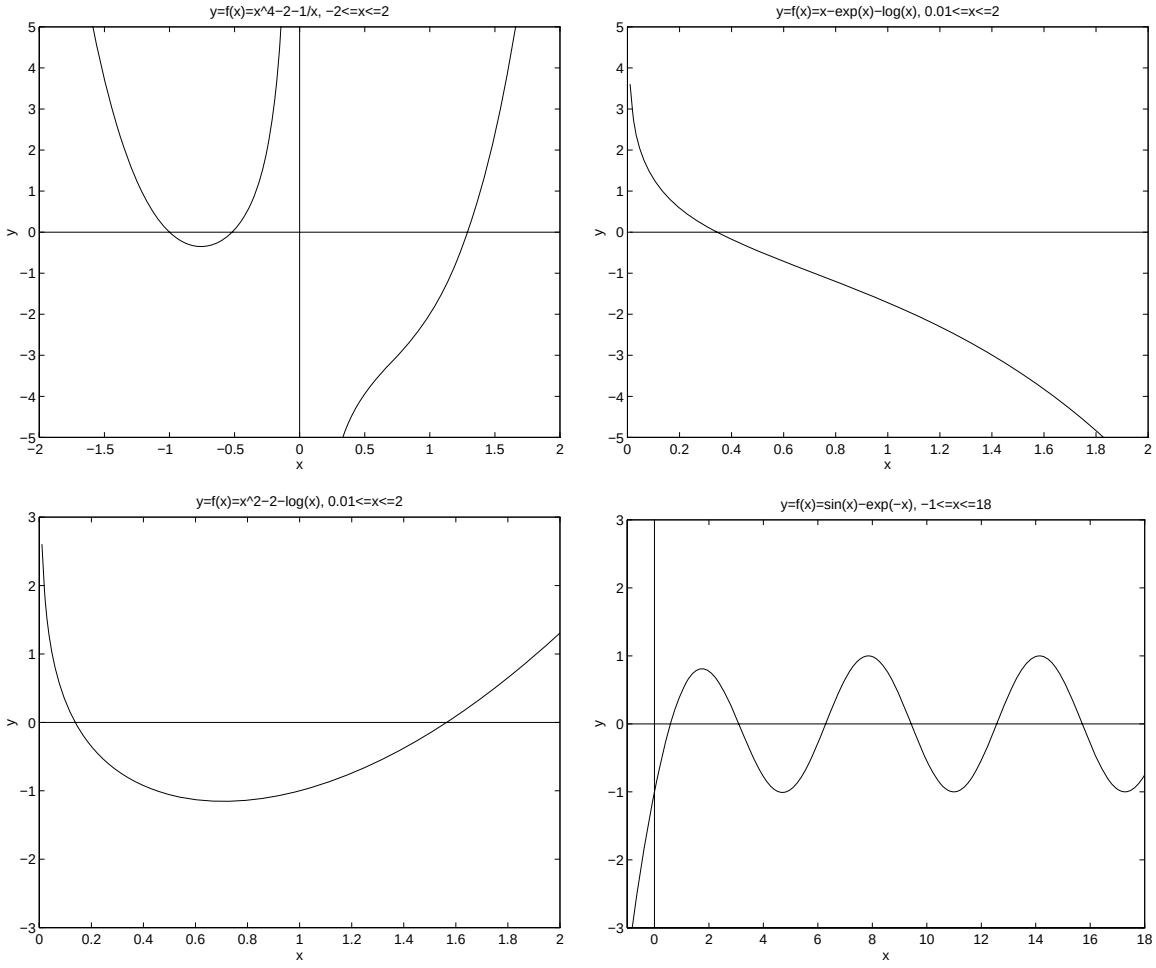

## 4.3 Weitere 2D/3D-Plots

Darstellungen von Funktionen im ebenen oder räumlichen Koordinatensystem mit Wahl des Darstellungsbereichs, Einzeichnen von Koordinatenachsen, Beschriftung.

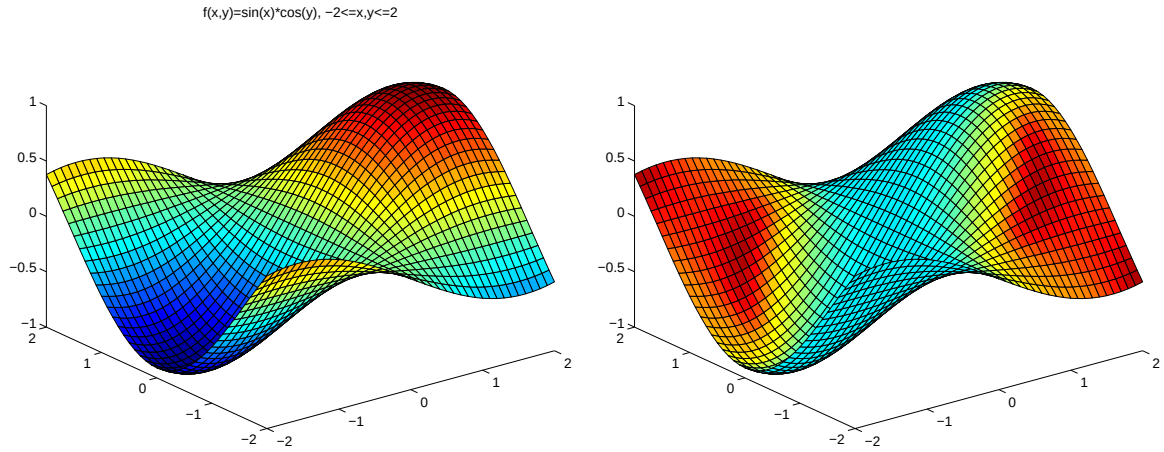
Die farbigen Darstellungen in MATLAB und den Postscript-Files stehen uns in dieser Arbeit nur als Druck in Schwarz-Weiß oder Graustufen zur Verfügung.

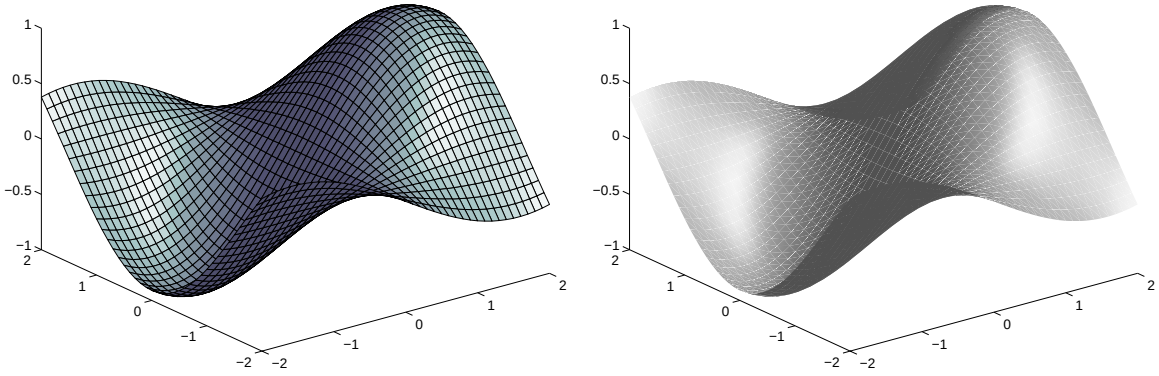
```
% 2D Plots
% 1. Beispiel
x = -2:0.01:2;
y = x.^4-2-1./x;
plot([x(1),x(max(size(x)))],[0,0],'w',...   % Fortsetzung des Kommandos
     [0,0],[-5,5],'w',x,y);
axis([-2 2 -5 5]);
title('y=f(x)=x^4-2-1/x, -2<=x<=2')
xlabel('x')
ylabel('y')
print graph301.ps -dps

% 2. Beispiel
x = 0.01:0.01:2;
y = x-exp(x)-log(x);
plot([x(1),x(max(size(x)))],[0,0],'w',x,y);
axis([0 2 -5 5]);
title('y=f(x)=x-exp(x)-log(x), 0.01<=x<=2')
xlabel('x')
ylabel('y')
print graph302.ps -dps
```

```
% 3. Beispiel
x = 0.01:0.01:2;
y = x.^2-2-log(x);
plot([x(1),x(max(size(x)))],[0,0],'w',x,y);
axis([0 2 -3 3]);
title('y=f(x)=x^2-2-log(x), 0.01<=x<=2')
xlabel('x');  ylabel('y')
print graph303.ps -dps

% 4. Beispiel
x = -1:0.1:18;
y = sin(x)-exp(-x);
plot([x(1),x(max(size(x)))],[0,0],'w',[0,0],[-3,3],'w',x,y);
axis([-1 18 -3 3]);
title('y=f(x)=sin(x)-exp(-x), -1<=x<=18')
xlabel('x');  ylabel('y')
print graph304.ps -dps
```



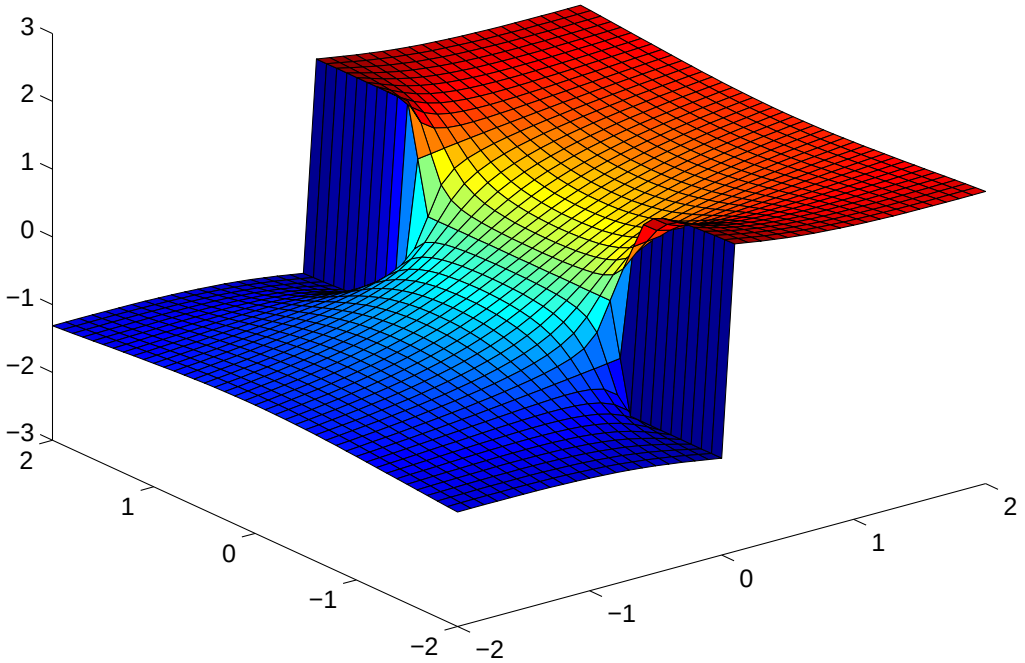
```
% 3D Plots
% 1. Beispiel
% f(x,y) = sin(x)*cos(y)
[x,y] = meshgrid(-2:0.1:2,-2:0.1:2);
z = sin(x).*cos(y);

% einfache Darstellung mit Gitter
surf(x,y,z);
shading faceted;
colormap(jet);
title('f(x,y)=sin(x)*cos(y), -2<=x,y<=2')
print graph401.ps -dps

% beleuchtete Darstellung mit Gitter
surfl(x,y,z);
shading faceted;
colormap(jet);
print graph402.ps -dps

% Graustufen mit Gitter
surfl(x,y,z);
colormap(bone);
print graph403.ps -dps

% interpolierte Graufarben ohne Gitter
shading interp;
print graph404.ps -dps
```



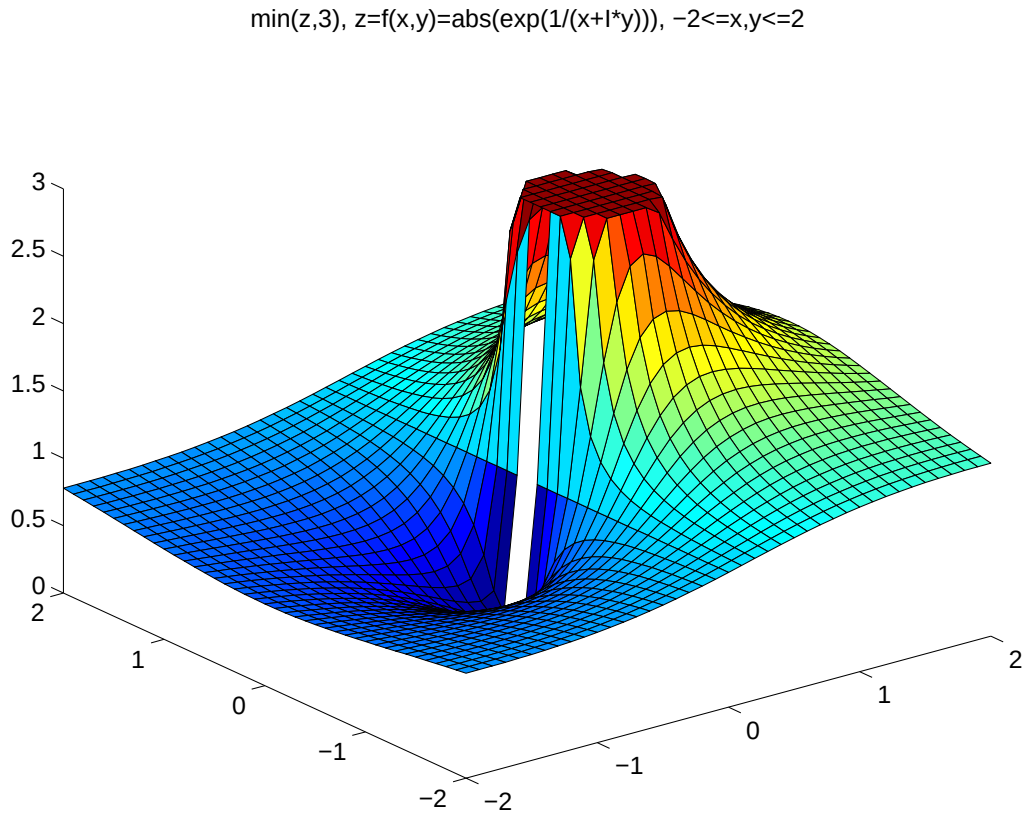


```
% 2. Beispiel
% f(x,y) = real(atan(x+I*y))
I = (-1)^(1/2);
[x,y] = meshgrid(-2:0.1:2,-2:0.1:2);
z = real(atan(x+I*y));
surf(x,y,z);
colormap(jet);
axis([-2 2 -2 2 -3 3]);
title('f(x,y)=real(atan(x+I*y)), -2<=x,y<=2')
print graph405.ps -dps
```

f(x,y)=real(atan(x+I*y)), −2<=x,y<=2



```
% 3. Beispiel
% f(x,y) = abs(exp(1./(x+I*y))
I = (-1)^(1/2);
[x,y] = meshgrid(-2:0.1:2,-2:0.1:2);
z = abs(exp(1./(x+I*y)));
surf(x,y,min(z,3));
colormap(jet);
axis([-2 2 -2 2 0 3]);
title('min(z,3), z=f(x,y)=abs(exp(1/(x+I*y))), -2<=x,y<=2')
print graph406.ps -dps
```




**Den Mehrfachplot in einer Abbildung haben wir schon verwendet.**

```
x = 0:0.01:2*pi;
y1 = sin(x); y2 = sin(2*x); y3 =sin(4*x);
plot(x,y1,x,y2,x,y3)
```

Die 3 Vektoren der Funktionswerte können auch zusammengefaßt werden zu einer Matrix, die dann im Plot-Befehl erscheint.

```
x = 0:0.01:2*pi;
Y = [sin(x)', sin(2*x)', sin(4*x)'];
plot(x,Y)
```

Weiterhin kann man mit dem `hold on`-Kommando eine Graphik "halten" und durch nachfolgende Graphikausgaben überlagern, bis die Freigabe durch `hold off` erfolgt.

# 5 Nichtlineare Gleichungen

## 5.1 Problemstellung und Iterationsverfahren (IV)

Zunächst die Darstellung einiger Grundlagen.

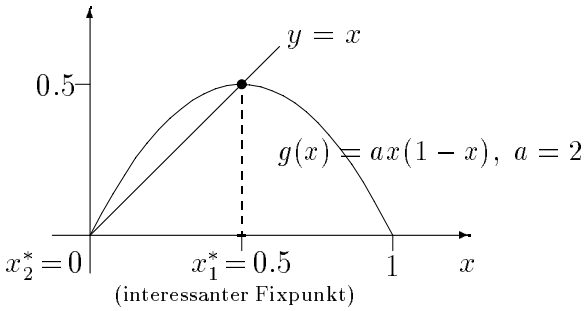
Nichtlineare Gleichung (Normalform, Nullstellenform)

$$f(x) \;=\; 0\;, \quad x^* \;-\; \text{Lösung (Nullstelle von } f).$$

Iterierfähige Form (Fixpunktform)

$$x \;=\; g(x)\;, \quad x^* \;-\; \text{Lösung (Fixpunkt von } g).$$

Illustration der Fixpunktsituation




### Äquivalenzaussage

1. Ist $x^*$ Fixpunkt von $g$, so ist $x^*$ Nullstelle von $f$ mit $f(x) = x - g(x)$.
2. Sei umgekehrt $x^*$ Nullstelle von $f$ und $\phi : \mathbb{R} \to \mathbb{R}$ eine Funktion mit $\phi(x) \neq 0$. Dann ist $x^*$ Fixpunkt von $g$ mit $g(x) = x - \phi(x)f(x)$.

### Allgemeines Iterationsverfahren (PICARD-Verfahren)

$$x_n = g(x_{n-1}), \quad n = 1, 2, 3, ..., \quad x_0 - \text{ Startnäherung.}$$

Der *Banachsche Fixpunktsatz* liefert Aussagen über Konvergenz, Grenzwert und Fehlerschätzungen.
Für den $n$-ten Iterationswert $x_n$ gilt mit $|g'(x)| \leq \lambda < 1$ die

- a-priori–Fehlerschätzung

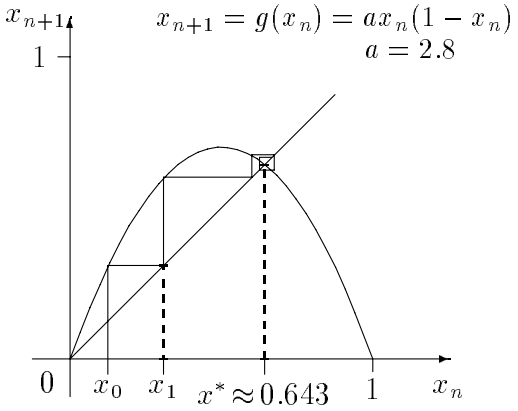
$$|x_n - x^*| \leq \frac{\lambda^n}{1-\lambda}|x_1 - x_0|,$$

- a-posteriori–Fehlerschätzung

$$|x_n - x^*| \leq \frac{\lambda}{1-\lambda}|x_n - x_{n-1}|.$$

Die erstere kann für eine grobe Abschätzung der Anzahl der notwendigen Iterationen bei gegebener Toleranz $\varepsilon$ gemäß $\frac{\lambda^n}{1-\lambda}|x_1 - x_0| \approx \varepsilon$ verwendet werden (Umstellung nach $n$). Des weiteren kann man die Konvergenzordnung des IV untersuchen.

**Illustration des Iterationsverfahren**




## 5.2 Ein- und zweistufige Iterationsverfahren

Allgemeine Form der einstufigen IV

$$x_n = x_{n-1} - \phi(x_{n-1})\ f(x_{n-1}), \quad n = 1, 2, 3, ...,$$

mit Startnäherung $x_0$.
Allgemeine Form der zweistufigen IV

$$x_n = x_{n-1} - \phi(x_{n-1}, x_{n-2})\ f(x_{n-1}), \quad n = 2, 3, 4, ...,$$

mit Startnäherungen $x_0,\ x_1$.

Einige IV sind nachfolgend aufgelistet, ohne aber auf die Angabe der Funktion $\phi(x)$, Voraussetzungen an das Verfahren und Startwert(e), Konvergenzfragen näher einzugehen.

**Ausgewählte ein- und zweistufige IV**

**Sehnenverfahren**

$$x_n = x_{n-1} - m \cdot f(x_{n-1}), \quad n = 1, 2, 3, ...,\ m = \text{const}$$

**Newtonverfahren, Tangentennäherungsverfahren (klassisches NV)**

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, \quad n = 1, 2, 3, ...$$

**Vereinfachtes Newtonverfahren**

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_0)}, \quad n = 1, 2, 3, ...$$

**Newtonähnliches Verfahren oder modifiziertes NV**

$$x_n = x_{n-1} - \frac{\Delta x}{f(x_{n-1} + \Delta x) - f(x_{n-1})}\, f(x_{n-1}), \quad n = 1, 2, 3, ...$$

**"Klassische" Regula falsi**

$$x_{n+1} = x_n - \frac{x_n - x_0}{f(x_n) - f(x_0)}\, f(x_n), \quad n = 1, 2, 3, ...$$

**"Verbesserte" Regula falsi, Sekantenverfahren**

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}\, f(x_n), \quad n = 1, 2, 3, ...$$

**Einschachtelungsverfahren**
- Regula falsi mit Intervallschachtelung
- Bisektionsverfahren, Verfahren der sukzessiven Intervallhalbierung

Die Rahmenbedingungen für IV sind

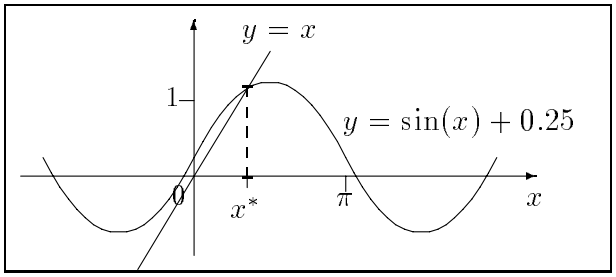
$$\begin{aligned} x_0\ (x_1) & \quad \text{- Startwert(e)} \\ n > n_{max},\ |f(x_n)| < \varepsilon, & \\ |x_n - x_{n-1}| < \varepsilon,\ \frac{|x_n - x_{n-1}|}{|x_n|} < \varepsilon,\ \frac{|x_n - x_{n-1}|}{1 + |x_n|} < \varepsilon & \quad \text{- Abbruchbedingungen} \\ n_{max} & \quad \text{- maximale Iterationszahl} \\ \varepsilon & \quad \text{- Toleranz} \end{aligned}$$

## 5.3 Das Newtonverfahren

Aufgabe ist die Bestimmung der reellen Nullstellen der Funktion $f(x) = x - \sin x - 0.25$ mit dem Newtonverfahren bei gegebener Abbruchbedingung.

**Graphische Lösung**

Zu Lokalisierung interpretieren wir die Nullstellen als Wurzeln der Gleichung $x = \sin x + 0.25 = g(x)$, also als Schnittpunkte der beiden Funktionen $y = x$ und $y = g(x)$.




Wertetafel für $g(x) = \sin x + 0.25$

| x | 0.9 | 1.0 | 1.1 | 1.2 | 1.3 | 1.5 |
|---|---|---|---|---|---|---|
| g(x) | 1.033 | 1.091 | 1.141 | 1.182 | 1.214 | 1.247 |

Daraus folgt für die einzige reelle Nullstelle $x^*$ die Abschätzung $1.1 < x^* < 1.3$, woraus als Startnäherung für das Iterationsverfahren der Wert $x_0 = 1.2$ genommen werden kann.

Die Implementierung in MATLAB soll auf der Struktur

**Rahmenprogramm** $\rightarrow$ {**NV, Tabelle**} $\rightarrow$ $\{f,\ f'\}$

basieren, wobei die einzelnen Komponenten als MATLAB-Funktionen in Form von entsprechenden $m$-Files bereitgestellt werden. Die Tabelle soll nach der Ermittlung der Nullstelle noch einmal einige Informationen über die Zwichenschritte des klassischen NV liefern und sein quadratisches Konvergenzverhalten zeigen. Der Bottom-up-Entwurf sieht folgende Dateien vor.

```
% f.m
function y = f(x)
y = x-sin(x)-0.25;

% fstrich.m
function y = fstrich(x)
y = 1-cos(x);

% newtonk.m
function [xsol,deltax,absf,n] = newtonk(f,fstrich,x0,tolx,tolf,nmax)
% Klassisches NV

k = 0;
xk = x0;
fk = feval(f,xk);
absfk = abs(fk);
xdiff = 2*tolx;

while k<nmax & absfk>tolf & abs(xdiff)>tolx,
  k = k+1;
  xdiff = fk/feval(fstrich,xk);
  xk = xk-xdiff;
  fk = feval(f,xk);
  absfk = abs(fk);
end;

xsol = xk;
deltax = abs(xdiff);
absf = absfk;
n = k;
% Ende der Funktion newtonk

% newtab.m
function tabelle = newtab(f,fstrich,x0,xstern,tolx,tolf,nmax)
```

```
% Zwischenschrittausgabe

k = 0;
xk = x0;
fk = feval(f,xk);
absfk = abs(fk);
xdelta = abs(xk-xstern);
xdiff = 2*tolx;
xdelalt = xdelta;

disp( '-------------------------------------------------');
disp([' k            x(k)  e(k)=|x(k)-x*|     e(k)/e(k-1)']);
disp( '-------------------------------------------------');
disp([sprintf('%2.0f',k),sprintf('%16.9f',xk),...
      sprintf('%16.4e',xdelta)]);

while k<nmax & absfk>tolf & abs(xdiff)>tolx,
  k = k+1;
  xdiff = fk/feval(fstrich,xk);
  xk1 = xk-xdiff;
  fk1 = feval(f,xk1);
  absfk = abs(fk1);
  xdelta = abs(xk1-xstern);
  quot = xdelta/xdelalt;

  disp([sprintf('%2.0f',k),sprintf('%16.9f',xk1),...
        sprintf('%16.4e',xdelta),sprintf('%16.4e',quot)]);

  xk = xk1;
  fk = fk1;
  xdelalt = xdelta;
end;
disp( '-------------------------------------------------');
% Ende der Funktion newtab
```

**Aufruf der Funktionen *newtonk* und *newtab* im Rahmenprogramm.**

```
% aufgabe.m
% Praktikumsaufgabe
% Klassisches Newtonverfahren
% Nullstelle und Tabelle Iterationswerte
% mit f, f', newtonk,newtab
clear
echo off
clc
diary protokol.txt
```

```
disp('Klassisches Newtonverfahren: Nullstelle und Tabelle Iterationswerte')

[xstern,deltx,fsol,n] = newtonk('f','fstrich',1.2,1.0e-8,1.0e-8,10)

newtab('f','fstrich',1.2,xstern,1.0e-8,1.0e-8,10);

diary off
pause
clc
```

Das führt zum Ergebnisprotokoll (ASCII-File) *protokol.txt*.

```
Klassisches Newtonverfahren: Nullstelle und Tabelle Iterationswerte

xstern =
   1.17122965250172

deltx =
    2.736020659854607e-007

fsol =
    3.452793606584236e-014

n =
     3

-------------------------------------------------
 k            x(k)  e(k)=|x(k)-x*|     e(k)/e(k-1)
-------------------------------------------------
 0     1.200000000     2.8770e-002
 1     1.171832302     6.0265e-004     2.0947e-002
 2     1.171229926     2.7360e-007     4.5400e-004
 3     1.171229653     0.0000e+000     0.0000e+000
-------------------------------------------------
```

Bei eingeschaltetem Birdschirmecho `echo on` werden in die Protokolldatei zusätzlich die Kommados

```
disp('Klassisches Newtonverfahren: Nullstelle und Tabelle Iterationswerte')

[xstern,deltx,fsol,n] = newtonk('f','fstrich',1.2,1.0e-8,1.0e-8,10)

newtab('f','fstrich',1.2,xstern,1.0e-8,1.0e-8,10);

diary off
```

geschrieben.

Bei der Implementierung des Newtonverfahrens wird oft das Parameterkonzept an die veränderten Bedingungen und gewünschten Ergebnisse angepaßt.
Die Bereitstellung der Ableitung der Funktion ersetzt man durch die Berechnung entsprechender Differenzenquotienten. Zwischenausgaben in der Funktion werden eingefügt.
Auf diese Weise können wir z.B. folgenden Varianten des NV erhalten.

```
% newton1.m
function x = newton1(x0,f,fs,tol,nmax)
% Klassisches Newtonsches Naeherungsverfahren fuer NLG
% mit Ausgaben

% Eingangsparameter
%  x0       Startwert
%  f        f(x)=0, Funktion als m-File
%  fs       f'(x),  Ableitung als m-File
%  tol      Toleranz
%  nmax     maximale Iterationsanzahl
% Ergebnisse
%  x        Loesung = Nullstelle oder letzte Iterierte
%  k        Anzahl der benoetigten Iterationen
%  |f(x)|   Funktionswert an der Loesung
%  Diverse Informationen ueber Iterationsverlauf

xn   = x0;  xnp1 = x0;
for k=1:nmax
  f1 = feval(f,xn);
  f2 = feval(fs,xn);
  xnp1 = xn-f1/f2;
  if (abs(xnp1-xn)<tol)
    disp(' ');
    disp(' Geforderte Genauigkeit erreicht!');
    disp(' Benoetigte Iterationen:');
    disp(k);
    break;
  end;
  xn = xnp1;
  if (k==nmax)
    disp(' ');
    disp('Maximale Iterationsanzahl erreicht!');
  end;
end;
x = xnp1;
disp(' ');
disp('f(x) = ');
disp(abs(feval(f,x)));
% Ende der Funktion newton1
```

```
% newton2.m
function x = newton2(x0,f,tol,nmax)
% Modifiziertes Newtonsches Naeherungsverfahren fuer NLG
% mit Ausgaben
%    f' ~ (f(x+h)-f(x))/h
.....................................................................

% newton3.m
function [xsol,deltax,deltaf,n] = newton3(f,x0,tolx,tolf,nmax)
% Modifiziertes Newtonsches Naeherungsverfahren fuer NLG
%     f' ~ (f(x+h)-f(x))/h

% Eingangsparameter
%  f        f(x)=0, Funktion als m-File
%  x0       Startwert
%  tolx     Toleranz fuer x, |x^(n)-x^(n-1)|<tolx
%  tolf     Toleranz fuer f, |f(x)|<tolf
%  nmax     maximale Iterationsanzahl
% Ergebnisse
%  xsol     Loesung = Nullstelle oder letzte Iterierte
%  deltax   |xsol-x^(n-1)|
%  deltaf   |f(xsol)|
%  n        Anzahl der benoetigten Iterationen
.....................................................................
```

Diese Funktionen werden in den nachfolgenden Beispielen verwendet.

## 5.4 Lösung einer Nullstellenaufgabe

Wir illustrieren die übliche Vorgehensweise anhand einer Aufgabe.

1. Ermittle die Anzahl der reellen Lösungen folgender Gleichungen, und gebe Intervalle für diese Lösungen an.

   (a) $x^4 - 2 = \frac{1}{x}$ (b) $x = e^x + \ln x$ (c) $x^2 - 2 = \ln x$
   (d) $\sin x = e^{-x}$ (e) $e^x - x - 1.01 = 0$ (f) $e^{-x} = x$

2. Bestimme die Lösung $x^*$ der Gleichung $e^{-x} = x$ mit dem allgemeinen Iterationsverfahren (AIV, Fixpunktiteration, Picard) $x_{n+1} = g(x_n)$ bis auf eine Genauigkeit von $\varepsilon = 10^{-6}$. Startwert sei $x_0 = 1$.

   - Graphische Lösung (siehe (f)).
   - Überprüfe die Voraussetzungen des Fixpunktsatzes, und gebe eine Iterationszahl $n$ zum Erreichen der Genauigkeit $\varepsilon$ an.
   - Gebe eine a-priori- und eine a-posteriori-Fehlerschätzung für den Fehler der Näherung $x_{10}$ an.

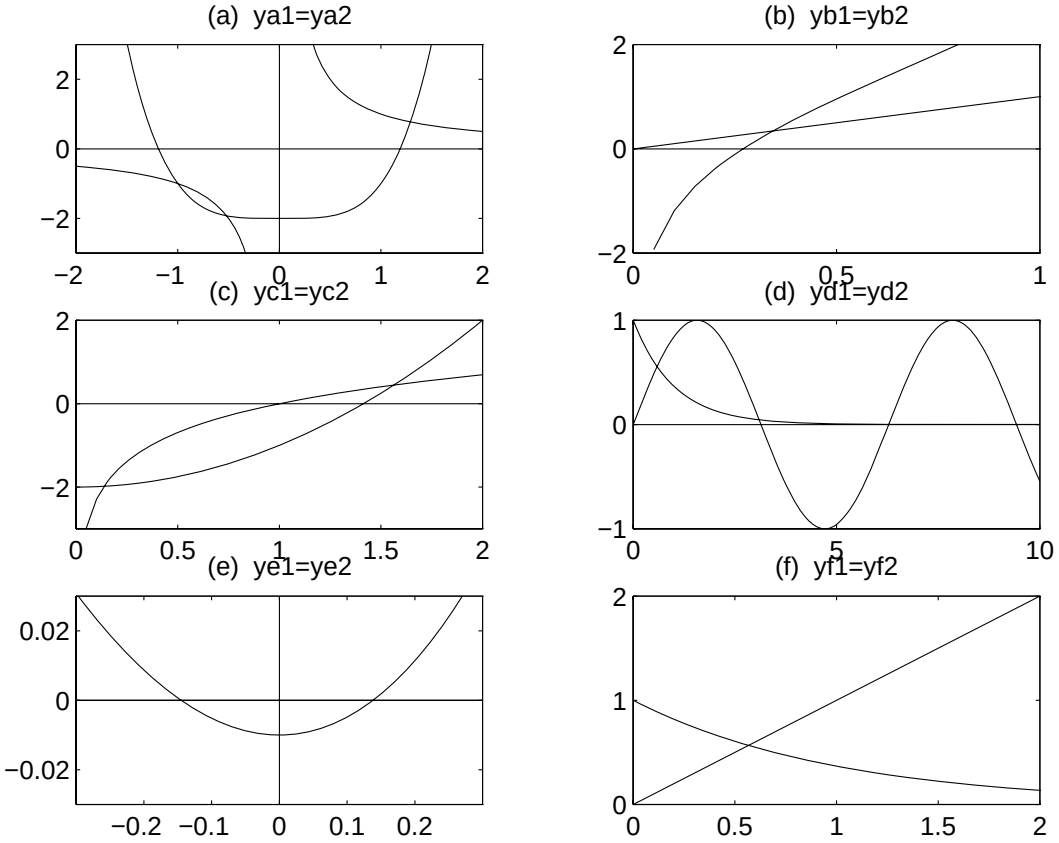
- Iteriere solange, bis die Lösung $x^*$ auf 3 Nachkommastellen genau bestimmt wurde.
- Notiere für die Werte $x_n$ den wahren Fehler $|x_n - x^*|$.
  Erläutere anhand der ermittelten wahren Fehler die *lineare* Konvergenz des Verfahrens.

3. Löse die Gleichung $e^{-x} = x$ mit derselben Startnäherung $x_0$ mit einer Genauigkeit von $10^{-6}$ mittels
   - klassischen NV,
   - modifizierten NV,
   - Regula falsi.

**Mögliche Implementierung in MATLAB.**

```
% Teil 1
x = linspace(0.0,4,80);
xachse = 0.*x;
% (a)
xx = -2:0.05:2;
xxachse = 0.*xx;
ya1 = xx.^4-2; ya2 = 1./xx;

Warning: Divide by zero
subplot(3,2,1)
plot(xx,xxachse,'w',xxachse,-4:0.1:4,'w',xx,ya1,'y',xx,ya2,'g')
axis([-2 2 -3 3])
title('(a)  ya1=ya2')
% (b)
yb1 = x; yb2 = exp(x)+log(x);

Warning: Log of zero
subplot(3,2,2)
plot(x,xachse,'w',x,yb1,'y',x,yb2,'g')
axis([0 1 -2 2])
title('(b)  yb1=yb2')
% (c)
yc1 = x.^2-2; yc2 = log(x);

Warning: Log of zero
subplot(3,2,3)
plot(x,xachse,'w',x,yc1,'y',x,yc2,'g')
axis([0 2 -3 2])
title('(c)  yc1=yc2')
% (d)
```

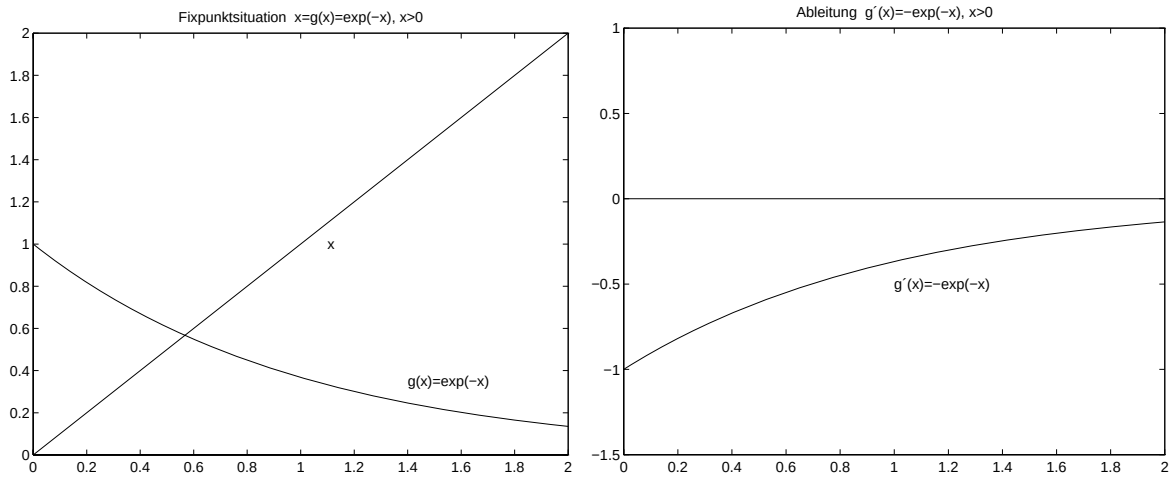
```
xx = 0:0.1:10;
xxachse = 0.*xx;
yd1 = sin(xx); yd2 = exp(-xx);
subplot(3,2,4)
plot(xx,xxachse,'w',xx,yd1,'y',xx,yd2,'g')
axis([0 10 -1 1])
title('(d)  yd1=yd2')
% (e)
xx = -0.3:0.01:0.3;
xxachse = 0.*xx;
ye1 = exp(xx)-xx-1.01; ye2 = 0.*xx;
subplot(3,2,5)
plot(xx,xxachse,'w',xxachse,xx,'w',xx,ye1,'y',xx,ye2,'g')
axis([-0.3 0.3 -0.03 0.03])
title('(e)  ye1=ye2')
% (f)
yf1 = exp(-x); yf2 = x;
subplot(3,2,6)
plot(x,xachse,'w',x,yf1,'y',x,yf2,'g')
axis([0 2 0 2])
title('(f)  yf1=yf2')
print ser3gr1.ps -dps
```



Auf die Fixpunktgleichung in (f) gehen wir nun etwas genauer ein.
Dazu werden einige vorbereitende Kommandos wie `diff, subs, numeric, num2str, isstr` gebraucht.

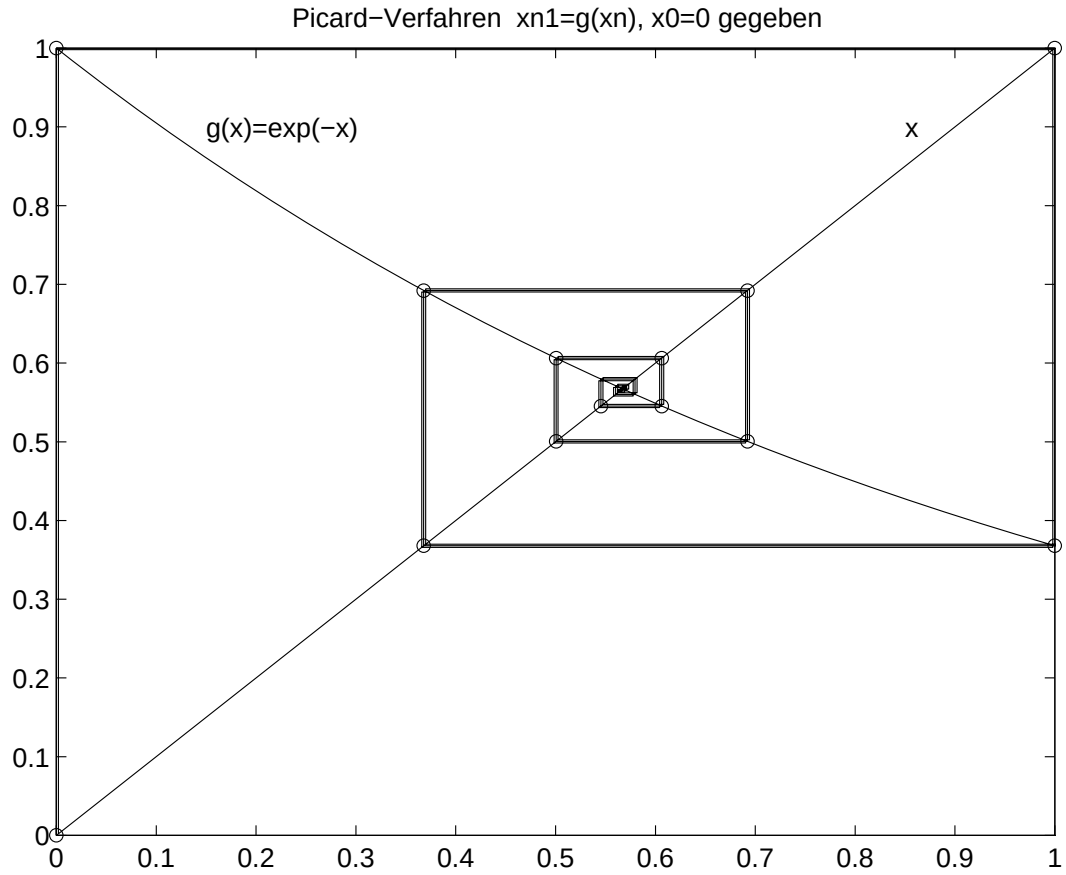
```
% Teil 2
% Reelle positive Loesung von exp(-x)=x und
% Ueberpruefung der Voraussetzungen des Fixpunktsatzes

x = linspace(0.0,2,80);
ya1 = exp(-x); ya2 = x;
plot(x,ya1,'y',x,ya2,'g')
axis([0 2 0 2])
title('Fixpunktsituation  x=g(x)=exp(-x), x>0')
text(1.1,1,'x')
text(1.4,0.35,'g(x)=exp(-x)')
print ser3gr2.ps -dps

clear x
disp('x>0')
x>0
disp('g(x) = exp(-x)')
g(x) = exp(-x)
g = 'exp(-x)'           % Einfache symbolische Funktionsdefinition

g =
exp(-x)

g(2);                   % Unsinn, da 2.Zeichen des String
x = 3;
gg = exp(-x)

gg =
    0.0498

g2s = subs(g,'2','x')   % g2s = subs(g,2,'x')

g2s =
exp(-2)

isstr(g2s)              % ist g2s ein String: ja=1

ans =
     1

g2n = numeric(g2s)      % numerische Auswertung eines String mit Zahl
```

```
g2n =
    0.1353

isstr(g2n)              % ist g2n ein String: nein=0

ans =
     0

% Ableitung berechnen und zeichnen

diff('exp(-x)','x');
g1 = diff('exp(-x)','x')   % symbolische Differentiation

g1 =
-exp(-x)

x = linspace(0.0,2,80);
for k=1:max(size(x))
  y1(k) = numeric(subs(g1,num2str(x(k)),'x'));
end;
plot(x,y1,'y',x,0*y1,'w')
axis([0 2 -1.5 1])
title('Ableitung  g'(x)=-exp(-x), x>0')
text(1.0,-0.5,'g'(x)=-exp(-x)')
print ser3gr3.ps -dps
```




Das AIV wird mit einem gegebenen Parametersatz durchgeführt.
Die Iterierten werden auf einem Vektor $xn$ abgespeichert, um dann den Ablauf der Iteration graphisch darzustellen. Insbesondere werden die beiden letzten Iterierten $x_k$, $x_{k-1}$ in der Gleitkommaarithmetik mit 16 Mantissenstellen gemerkt (Kommando `vpa`, variable precision floating-point arithmetic with $d$ decimal digits, $d = 16$ default), um sie bei den Fehlerabschätzungen zu verwenden.

```
% Fixpunktiteration, Picard-Verfahren

x0 = 0;
xn(1) = x0;
ndiff = 1;
k = 0;
maxit = 30;
epsi = 1e-6;
while (k<maxit) & (ndiff>epsi)
  k = k+1;
  xk = xn(k);
  xn(k+1) = exp(-xk);
  ndiff = abs(xn(k+1)-xk);
end;
k = k+1

k =
     27

xkm1 = vpa(xk)

xkm1 =
.5671437480994115

xk = vpa(xn(k))

xk =
.5671430308342420

% Umspeichern von xn auf Polygonzug

for l=1:k-1
  xng(2*l-1)=xn(l); xng(2*l)=xn(l);
  yng(2*l-1)=xn(l); yng(2*l)=xn(l+1);
end

plot(x,ya1,'y',x,ya2,'g',xng,yng,'r',xng(1:k/2),yng(1:k/2),'ro')
axis([0 1 0 1])
title('Picard-Verfahren  xn1=g(xn), x0=0 gegeben')
text(0.85,0.9,'x')
text(0.15,0.9,'g(x)=exp(-x)')
print ser3gr4.ps -dps
```




Das AIV liefert $|x_{27}-x_{26}| \leq 10^{-6}$. Kontrollieren wir die Güte der a-priori–Fehlerschätzung

$$|x_k - x^*| \leq \frac{\lambda^k}{1-\lambda}|x_1 - x_0| \leq 10^{-6},$$

die im rechten Teil nach $k$ umgestellt werden kann.

$$k \approx \frac{\log \frac{10^{-6}(1-\lambda)}{|x_1-x_0|}}{\log \lambda}, \quad \lambda \approx |g'(x_{27})|.$$

Es ist eine gute Annäherung zu erkennen.

```
% Iterationsanzahl abschaetzen
lambdak = abs(-exp(-xn(k)));
niter = ceil(log(epsi*(1-lambdak)/abs(xn(2)-xn(1)))/log(lambdak));
sprintf('Iterationsanzahl  niter = %3.0f',niter)

ans =
Iterationsanzahl  niter =  26
```

Nun nehmen wir die Fehlerschätzungen für die Näherung $x_{10}$ mit den gespeicherten Iterierten vor. Dazu brauchen wir noch den Grenzwert $x^*$ mit maximaler Genauigkeit.

```
% Schaetzungen der 10.Naeherung

x9  = xn(10);
x10 = xn(11);
xexakt = 0.567143290409784;
vpa(x9,7)

ans =
.5711431

vpa(x10,7)

ans =
.5648793

vpa(xexakt,16)

ans =
.5671432904097840

format long;
lambda = abs(-exp(-xexakt))

lambda =
   0.56714329040978

k = 10;
% a priori-Abschaetzung
w = lambda^k/(1-lambda)*abs(xn(2)-xn(1));
disp('Abschaetzungen')
disp('a priori:')
sprintf('|x10-x*| = %9.7f <= %9.7f  = lambda^k/(1-lambda)|x1-x0|',...
         abs(x10-xexakt),w)

ans =
|x10-x*| = 0.0022639 <= 0.0079540  = lambda^k/(1-lambda)|x1-x0|

% a posteriori-Abschaetzung
w = lambda/(1-lambda)*abs(x10-x9);
disp('a posteriori:')
sprintf('|x10-x*| = %9.7f <= %9.7f = lambda/(1-lambda)|x10-x9|',...
         abs(x10-xexakt),w)

ans =
|x10-x*| = 0.0022639 <= 0.0082070 = lambda/(1-lambda)|x10-x9|
```

```
format short;
disp('Lineare Konvergenz')
disp('k  q=e(k+1)/e(k)   |x(k)-xexakt|')
disp('-------------------------------------')
q = 1;
for l=1:k+4
  s1 = num2str(l);
  s2 = num2str(abs(xn(l)-xexakt));
  s3 = num2str(abs(xn(l)-xexakt)/q);
  q = abs(xn(l)-xexakt);
  if l<10
    s12 = [' ',s1,'    ',s3,'        ',s2];
   else
    s12 = [s1,'    ',s3,'        ',s2];
  end;
  disp(s12)
end
```

```
Lineare Konvergenz
k  q=e(k+1)/e(k)   |x(k)-xexakt|
-------------------------------------
 1    0.5671        0.5671
 2    0.7632        0.4329
 3    0.4603        0.1993
 4    0.6276        0.1251
 5    0.5331        0.06667
 6    0.5865        0.0391
 7    0.5562        0.02175
 8    0.5734        0.01247
 9    0.5636        0.007028
10    0.5691        0.004
11    0.5660        0.002264
12    0.5678        0.001285
13    0.5668        0.0007286
14    0.5673        0.0004133
```

Beide Fehlerschätzer unterscheiden sich nicht wesentlich. Wir haben eine lineare Konvergenzordnung mit dem Konvergenzfaktor $q \approx \lambda = |g'(x^*)| \approx 0.5671$.

Zum Vergleich rechnen wir 3 andere Iterationsverfahren für die zugehörige Nullstellengleichung $f(x) = e^{-x} - x = 0$.

```
% Teil 3
% Nullstellenverfahren fuer skalare Gleichungen f(x)=0
% Verfahren als m-Files
```

```
x0 = 0;
x1 = 1;
TOL = 1e-7;
MaxIt = 10;
format long;
% klassisches NV
xs = newton1(x0,'f2','f2s',TOL,MaxIt)  % m-Files

Geforderte Genauigkeit erreicht!
Benoetigte Iterationen:
     5

f(x) =
    1.110223024625156e-016

xs =
   0.56714329040978

% modifiziertes NV
disp('Modifiziertes NV mit  f'(x) ~ (f(x+h)-f(x))/h')
xs = newton2(x0,'f2',TOL,MaxIt)        % m-File

Geforderte Genauigkeit erreicht!
Benoetigte Iterationen:
     5

f(x) =
     0

xs =
   0.56714329040978

% Regula falsi
xs = regula(x0,x1,'f2',TOL,MaxIt)      % m-File

Geforderte Genauigkeit erreicht!
Benoetigte Iterationen:
     5

f(x) =
    1.242339564555550e-013

xs =
   0.56714329040970
```

## 5.5 Bestimmung des Kehrwerts einer Zahl

Es soll der Kehrwert einer gegebenen reellen Zahl $a > 0$ ohne die Benutzung von Divisionen berechnet werden.

1. Man leite dazu die Form des Newtonverfahrens für die Nullstellenaufgabe $f(x) = \frac{1}{x} - a = 0$ her.

2. Man ermittle den Konvergenzbereich des Newtonverfahrens von Abhänigkeit vom Parameter $a$ und geeignete Startwerte $x_0$ (ohne Divisionen) für die Iteration.

3. Man bestimme den Kehrwert von $a = 2$ mit verschiedenen Startwerten bis auf eine Genauigkeit von $\varepsilon = 10^{-6}$.

Schwerpunkt legen wir hier auf die Benutzung von weiteren symbolischen Berechnungen. Das erste Kommando `fzero` bezieht sich auf die Bestimmung der Nullstelle einer skalaren Funktion nahe einer gewünschten Stelle und mit geforderter Toleranz (default `eps`). Die Funktion muß als $m$-File gegeben sein oder eine Standardfunktion sein.
Der zweite Aspekt ist die Verwendung des Newtonverfahrens mit ihren Funktionen $f$, $f'$ als Parameter. Es ist nicht möglich, den Parameter $a$ in diesen Funktionen als globale Größe zu benutzen.
Die zur Verfügung stehenden $m$-Files für Funktionen sind

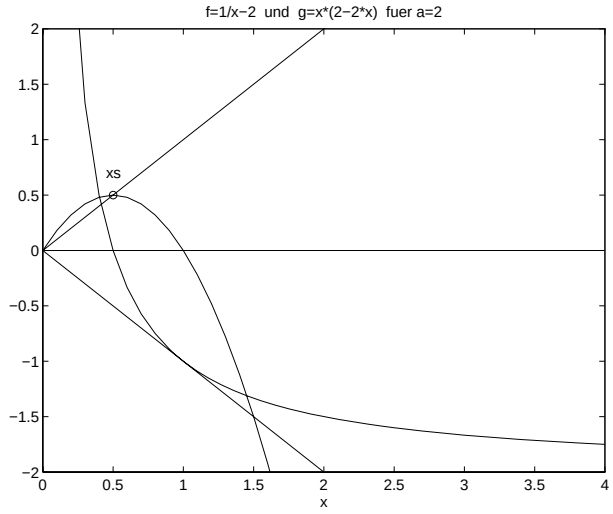
```
% f3.m
function y = f3(x)
  y = 1/x-2;     % a=2
% f3s.m
% Ableitung von f3
function y = f3s(x)
  y = -1/(x^2);

% f4.m  mit Parameter a
function y = f4(x,a)
  y = 1/x-a;
% f4s.m
% Ableitung von f4
function y = f4s(x,a)
  y = -1/(x^2);
```

### Bestimmung des Kehrwerts mit `fzero` und NV.

Das NV basiert auf der Fixpunktgleichung $x = g(x)$, $g(x) = x(2 - 2x)$, $a = 2$. Der Fixpunkt ist dann $x^* = 0.5$.

```
% Teil 1
a = 2;
f = '1./x-2';
fplot(f,[0.1 3])
title(f);  xlabel('x')
```

```
xh = 0:0.1:4;
fh = 1./xh-2;
gh = xh.*(2-2*xh);
plot(xh,xh,'y',xh,-xh,'y',xh,fh,'r',xh,gh,'g',xh,0*xh,'w',...
     0.5,0.5,'ro')
axis([0 4 -2 2])
title('f=1/x-2  und  g=x*(2-2*x)  fuer a=2')
xlabel('x')
print ser3gr5.ps -dps   % Warning: Divide by zero
```

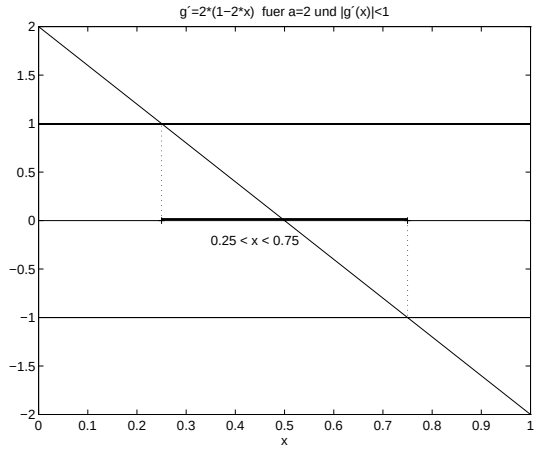



```
disp('UP fzero')
xs = fzero('f3',0.4)    % NS von f3=1/x-2 nahe 0.4

xs =
    0.5000

TOL = 1e-7;  x0 = 0.4;  MaxIt = 10;
disp('NV')
% xs = newton1(x0,'f4','f4s',TOL,MaxIt)
% geht nicht wegen Parametr a in f4
xs = newton1(x0,'f3','f3s',TOL,MaxIt)    % m-File

Geforderte Genauigkeit erreicht!
Benoetigte Iterationen:
     5

f(x) =
     0

xs =
    0.5000
```

Für die Konvergenzbetrachtungen nehmen wir die Ableitung $g'(x)$ und schätzen ihren Betrag im Intervall [0,1] ab.
Das Kommando, welches ein lokales Minimum einer Funktion $f(x)$ im gegebenen Intervall finden kann, ist `fmin`. Soll das Maximum gefunden werden, wird die Funktion $-f(x)$ betrachtet. Die Verkettung von Strings führt das Kommando `symop` (symbolic operations) aus.

```
% Teil 2  Konvergenzbereich
g = 'x*(2-2*x)'
gs = diff(g,'x')      % symbolische Differentiation

gs =
2-4*x

fplot(gs,[0 1])       % Pretty Plot
title('g'(x) fuer a=2')

xmin = fmin(gs,0,1)

xmin =
    0.9999

gss = symop('-',gs)

gss =
-2+4*x

xmax = fmin(gss,0,1)

xmax =
  6.6107e-005

subs(gs,xmin,'x');    % ans = -2251502094398737/1125899906842624

numeric(subs(gs,xmin,'x'))

ans =
   -1.9997

numeric(subs(gs,xmax,'x'))

ans =
    1.9997

xh = 0:0.1:1;
gh = 2*(1-2*xh);
```

```
plot(xh,0*xh,'w',xh,ones(max(size(xh))),'y',...
     xh,-ones(max(size(xh))),'y',xh,gh,'g',...
     0.25,0,'w+',0.75,0,'w+',...
     [0.25,0.75],[0.01,0.01],'w',[0.25,0.75],[0.02,0.02],'w',...
     [0.25,0.25],[0,1],':',[0.75,0.75],[0,-1],':')
axis([0 1 -2 2])
title('g'=2*(1-2*x)  fuer a=2 und |g'(x)|<1')
text(0.35,-0.2,'0.25 < x < 0.75')
xlabel('x')
print ser3gr6.ps -dps

disp('|g'(x)|<1 gdw. 1/(2a)< x < 3/(2a)')
disp('Startwerte x0 fuer NV zulaessig aus (0,1)')
```




Der eigentliche Argumentbereich mit $|g'(x)| < 1$, $a = 2$, ist $x \in (\frac{1}{4}, \frac{3}{4})$. Dieser kann jedoch auf das größere Intervall (0,1) ausgedehnt werden, denn der Fixpunkt $x^* = 0.5$ ist dort immer anziehend.

```
% Teil 3  Kehrwert mit verschiedenen Startwerten
a = 2;
TOL = 1e-4;
for k=1:7
  x0 = k/8;
  MaxIt = 10;
  x2s(k) = newton1(x0,'f3','f3s',TOL,MaxIt);
end;
disp('Klassisches NV fuer a=2')
disp('x0      xsk   ')
disp('----------------------')
for k=1:7
  x0 = k/8;
  s1 = sprintf('%5.3f',x0);
  s2 = sprintf('%10.8f',x2s(k));
  s12 = [s1,'   ',s2];
  disp(s12)
end;
```

Die Genauigkeit $10^{-4}$ wird im Rahmen der maximalen Iterationsanzahl 10 nach wenigen Iterationen erreicht. Für die Startwerte $x_{0,k} = \frac{k}{8}$ braucht das NV entsprechend $n_k = 6, 5, 4, 1, 4, 5, 6$ Iterationen, wobei für $x_{0,4} = \frac{1}{2} = x^*$ natürlich nur ein Iterationsschritt auszuführen ist.
Die Funktionswerte $f(x) = 1/x - 2$ an der letzten Iterierten liegen in der Größenordnung $\mathcal{O}(10^{-8}..10^{-10})$.

```
Klassisches NV fuer a=2
x0      xsk
---------------------
0.125   0.49999999495
0.250   0.49999999988
0.375   0.49999999988
0.500   0.50000000000
0.625   0.49999999988
0.750   0.49999999988
0.875   0.49999999495
```

## 5.6 Parameterkonzept von modifizierten NV, NV und Regula falsi

Wir hatten schon die Lösung von $e^{-x} = x$ mit dem modifizierten NV *newton*2 berechnet. Günstiger ist es, die meist zahlreichen Ergebnisse einer Funktion als Parameterliste zu übergeben. Deshalb soll hier abschließend die Funktion *newton*3, deren Parameter weiter oben beschrieben sind, angewendet werden.

```
% f2.m
function y = f2(x)
  y = exp(-x)-x;


% m-File newton3 fuer NV

[xsol,deltax,deltaf,n] = newton3('f2',0.5,1e-6,1e-6,10);

disp('Ergebnisse ');
s1 = num2str(xsol);
disp(['xsol = ',s1]);
s1 = num2str(deltax);
disp(['deltax = ',s1]);
s1 = num2str(deltaf);
disp(['deltaf = ',s1]);
s1 = num2str(n);
disp(['n = ',s1]);
```

```
Ergebnisse
xsol = 0.5671
deltax = 0.000832
deltaf = 1.941e-007
n = 2
```

Weiterhin taucht die Frage auf, ob man nicht direkt in die Parameterliste der Funktionen die Zeichenkette mit dem Funktionsausdruck (rechte Seite) eintragen kann. Das geht, aber ist beschränkt auf die alleinige Angabe **eines Standardfunktionsnamens**.

```
% f.m
function y = f(x)
  y = x-sin(x)-0.25;

% fstrich.m
function ystrich = fstrich(x)
  ystrich = 1-cos(x);

% Regula falsi
x0 = 0.5; x1 = 1.5;
xs = regula(x0,x1,'f',1e-6,100)

Geforderte Genauigkeit erreicht!
Benoetigte Iterationen:
     7

f(x) =
  2.2204e-016

xs =
    1.1712

% Parameter = einfache Standardfunktion
x0 = 2; x1 = 4;
xs = regula(x0,x1,'sin',1e-6,100)

Geforderte Genauigkeit erreicht!
Benoetigte Iterationen:
     5

f(x) =
  1.2246e-016

xs =
    3.1416
```

```
% Nicht zulaessig
% Parameter = Ausdruck mit Standardfunktion
x0 = 1.5; x1 = 2;
xs = regula(x0,x1,'sin+cos',1e-6,100)
xs = regula(x0,x1,'0.1+cos',1e-6,100)
xs = regula(x0,x1,'f+cos',1e-6,100)

???Error using ==> feval
Undefined function sin+cos, 0.1+cos bzw. f+cos.


% Klassisches NV
[xsol,deltax,absf,n] = newtonk('f','fstrich',1,1e-6,1e-6,10)

xsol = 1.1712
deltax = 5.6115e-004
absf = 1.4507e-007
n = 3

[xsol,deltax,absf,n] = newtonk('sin','cos',3,1e-6,1e-6,10)

xsol = 3.1416
deltax = 9.5389e-004
absf = 2.8932e-010
n = 2

% Nicht zulaessig
% Parameter = Ausdruck mit Standardfunktion
[xsol,deltax,absf,n] = newtonk('sin+cos','cos-sin',3,1e-6,1e-6,10)

??? Error using ==> feval
Undefined function sin+cos.
```

# 6 Nichtlineare Gleichungssysteme

## 6.1 Problemstellung und Iterationsverfahren

Auch hier die Darstellung einiger Grundlagen.

Fixpunktform in vektorieller Darstellung

$$x = g(x), \quad g : \mathbb{R}^n \to \mathbb{R}^n.$$

**Allgemeines Iterationsverfahren (PICARD-Verfahren)**

$$x^{(k)} = g(x^{(k-1)}), \quad k = 1, 2, 3, ..., \text{ Startnäherung } \; x^{(0)} \in \mathbb{R}^n.$$

**Konvergenz des Iterationsverfahrens**

Das Verfahren konvergiert gegen einen Fixpunkt $x^*$ von $g$ in einer ausgewählten Norm, falls

$$\lim_{k\to\infty} \|x^{(k)} - x^*\| = 0, \text{ anders notiert als } \lim_{k\to\infty} x^{(k)} = x^*.$$

Es gibt einen *Fixpunktsatz* für Systeme mit entsprechenden Fehlerschätzungen.
Für den $k$-ten Iterationsvektor $x^{(k)}$ gilt mit

$$S = \{x| \; \|x - x^{(0)}\| \le \varrho\}, \quad \|g(x^{(0)}) - x^{(0)}\| \le (1-\lambda)\varrho,$$

$$\|g(x) - g(y)\| \le \lambda\|x - y\| \quad \forall \; x, y \in S, \quad 0 \le \lambda < 1,$$

die

- a-priori–Fehlerschätzung

$$\|x^{(k)} - x^*\| \le \frac{\lambda^k}{1-\lambda}\|x^{(1)} - x^{(0)}\|,$$

- a-posteriori–Fehlerschätzung

$$\|x^{(k)} - x^*\| \le \frac{\lambda}{1-\lambda}\|x^{(k)} - x^{(k-1)}\|.$$

Dabei spielen für die Konvergenz die *Jacobimatrix* von $g$

$$G(x) = (g_{ij}(x)) = \left(\frac{\partial g_i(x)}{\partial x_j}\right)$$

sowie die Kontraktionsbedingung in einer Matrixnorm

$$\|G(x)\| \le \lambda < 1 \quad \forall \; x \in S$$

eine wichtige Rolle.

## 6.2 Newtonverfahren für Gleichungssysteme

Das NV nimmt eine herausragende Stellung unter den IV ein.
Nichtlineares Gleichungssystem (Normalform, Nullstellenform)

$$f(x) = 0, \quad f : \mathbb{R}^n \to \mathbb{R}^n, \quad x^* \; - \; \text{Lösungsvektor.}$$

Allgemeine Form der IV

$$x^{(k)} = x^{(k-1)} - \Phi(x^{(k-1)})\; f(x^{(k-1)}), \quad k = 1, 2, 3, ...$$

mit Startvektor $x^{(0)}$ und Matrix $\Phi(x)$.

Mit der *Jacobimatrix* von $f$

$$F(x) = f'(x) = (f_{ij}) = \left(\frac{\partial f_i}{\partial x_j}\right) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}$$

trifft man für die Matrix $\Phi(x)$ die Wahl

$$\Phi(x) = [F(x)]^{-1} = [f'(x)]^{-1}.$$

**Newtonverfahren**
Die elegante Notation ist die als lineares Gleichungssystem.

$$\begin{aligned} F(x^{(k)})\; \delta^{(k)} &= f(x^{(k)}) \\ x^{(k+1)} &= x^{(k)} - \delta^{(k)}, \quad k = 0, 1, 2, ... \end{aligned}$$

Die Rahmenbedingungen für das Verfahren sind

$$\begin{aligned} x^{(0)} & \quad \text{- Startvektor} \\ k > k_{max}, \; \|\delta^{(k)})\| < \varepsilon \;\text{ oder }\; \|f(x^{(k)})\| < \varepsilon & \quad \text{- Abbruchbedingungen} \\ k_{max} & \quad \text{- maximale Iterationszahl} \\ \varepsilon & \quad \text{- Toleranz} \end{aligned}$$

Eine Reihe von Modifikationen, insbesondere in der Ersetzung der Jacobimatrix durch geeignete einfachere Matrizen oder Näherungen, sind üblich.
Die Konvergenz wird mit lokalen oder semilokalen Konvergenzsätzen beschrieben. Dabei sind die Glattheit der Funktion und die Regularität der Jacobimatrix an der lokalen Lösung entscheidende Aspekte.

Die partiellen Ableitungen $\frac{\partial f_i}{\partial x_j}$ von $f(x)$ werden oft näherungsweise mittels einfacher Differenzenquotienten bestimmt.

$$\frac{\partial f_i}{\partial x_j} \approx \frac{f_i(x_1, x_2, ..., x_{j-1}, x_j + h_j, x_{j+1}, ..., x_n) - f_i(x_1, ..., x_n)}{h_j},$$

wobei $h_j \approx \varepsilon \|f(x^{(0)})\|$ ist. Der Faktor $\varepsilon$ korrespondiert mit der halben Mantissenlänge der zugrundeliegenden Gleitkommaarithmetik. Oft werden die Schrittweiten $h_j$ in den Koordinatenrichtungen gleich gewählt.
Das Parameterkonzept entsprechender Routinen enthält zumeist verschiedene Toleranzen für die Abbruchbedingungen, Ergebnisparameter, die den Ablauf des Verfahrens beschreiben (auch verbale), oder Kontrollgrößen wie Anzahl der Funktionsaufrufe von $f$, Test der Jacobimatrix bzw. genäherten Jacobimatrix auf Singularität sowie $k$, $(|\delta_1^{(k)}|, |\delta_2^{(k)}|, ..., |\delta_n^{(k)}|)$, $\|f(x^{(k)})\|$, $\|x^{(k+1)} - x^{(k)}\|$.

```
% newtons.m
function [xs,Defectx,Defectf,Error1,ItNr] = newtons(x0,f,Tolx,Tolf,MaxIt);

 % Modified Newton method for nonlinear systems

   h      = 1e-5;          % for approximation of derivative
   n      = max(size(x0));
   ItNr   = 0;
   Error1 = 'No errors';
   xk     = x0;

   DF = zeros(n,n);
   Fk = feval(f,xk);       % evaluation of function f like parameter
   Defectf = norm(Fk,2);
   Defectx = 0;

   for k=1:MaxIt
     if (Defectf<Tolf)
       Error1 = 'Small norm value of function';
       xs = xk;
       return;
     end;
     % Evaluate Jacobian without its control of regularity
     for j=1:n
       e = zeros(n,1);
       e(j) = 1.0;
       DF(1:n,j) = (1/h)*(feval(f,xk+h*e)-Fk);
     end;

     % Compute ordinary Newton correction
     dxk = DF\Fk;
```

```
    % Compute trial iterate
    xkp1 = xk-dxk;
    ItNr = ItNr+1;
    Defectx = abs(dxk);
    % Evaluate function
    Fkp1 = feval(f,xkp1);
    if (norm(dxk,2)<Tolx)
      Error1 = 'No further change of solution';
      xs = xk;
      return;
    end;

    xk = xkp1;
    Fk = Fkp1;
    Defectf = norm(Fk,2);
  end;
  Error = 'Too many iterations needed';
  xs    = xk;
% end of function newtons
```

## 6.3 Fixpunktiteration

Gegeben ist das nichtlineare Gleichungssystem in iterierfähiger Form (Fixpunktform).

$$\begin{aligned} x_1 &= \frac{1}{4}(x_1 \sin x_2 + x_2) = g_1(x_1, x_2), \\ x_2 &= \arctan \frac{4}{x_1 + x_2} = g_2(x_1, x_2). \end{aligned}$$

1. Notiere das allgemeine Iterationsverfahren und veranschauliche, daß $x^{(0)} = (0.3, 1.0)^T$ eine geeignete Startlösung darstellt.

2. Überprüfe numerisch, ob für $x^{(0)}$ die Voraussetzungen des Fixpunktsatzes (Konvergenzkriterien) erfüllt sind.

3. Berechne die Näherungen $x^{(1)}, x^{(2)}, ..., x^{(8)}$ mit dem allgemeinen Iterationsverfahren, und gebe eine a-posteriori-Schätzung des Fehlers $\|x^{(8)} - x^*\|_\infty$ an.

Zunächst iterieren wir einfach das AIV bei gegebenem Startpunkt.
Die rechte Seite des Systems stellen wir als $m$-File bereit. Das Verfahren konvergiert gegen den Fixpunkt $x^* = (0.3888999487733591, 1.1939841545645480)$.

```
% g6.m
function y = g6(x)
  y = zeros(min(size(x)):max(size(x)))';
  y(1) = 0.25*(x(1)*sin(x(2))+x(2));
  y(2) = atan(4/(x(1)+x(2)));
```

```
% Teil 1  Fixpunktiteration
x0 = [0.3 1.0]';
xk1 = x0;
deltax = 1;
k = 0;
maxit = 10;
epsi = 1e-6;
while (k<maxit) & (deltax>epsi)
  k = k+1;
  xk = xk1;
  xk1 = g6(xk);     %  m-File
  deltax = norm(xk1-xk,2);
end;

disp('Ergebnisse ');
s1 = sprintf('%9.6f',xk(1));
s2 = sprintf('%9.6f',xk1(2));
disp(['xk(1,2) = (',s1,',',s2,')']);
s1 = num2str(deltax);
disp(['deltax = ',s1]);
s1 = num2str(k);
disp(['k = ',s1]);

Ergebnisse
xk(1,2) = ( 0.388900, 1.193984)
deltax = 7.502e-007
k = 7
```

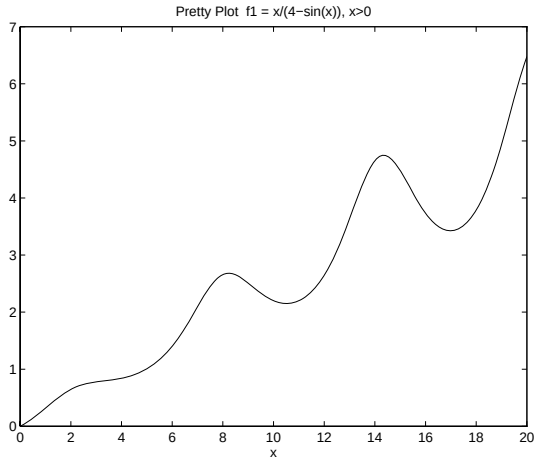
Nun verschaffen wir uns einen Überblick über die Lösungsmenge. Dazu stellen wir beide Gleichungen um nach $x_1$.

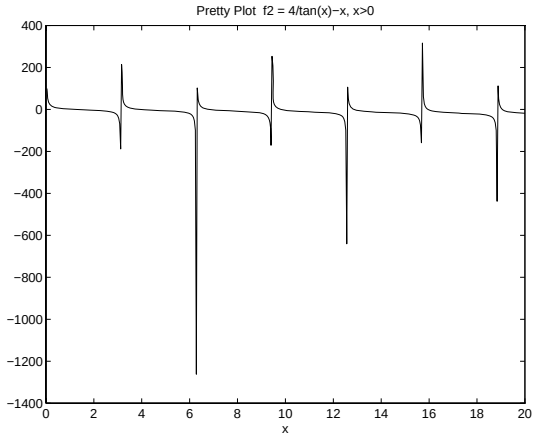
$$\begin{aligned} x_1 &= \frac{x_2}{4-\sin x_2}, \\ x_1 &= \frac{4}{\tan x_2} - x_2, \end{aligned}$$

so daß man nun die Schnittpunkte von zwei Kurven im Koordinatensystem $(x_2, x_1) = (x, y)$ betrachten kann.
Wir plotten zunächst die Kurven einzeln, wobei der Pretty Plot der zweiten wegen der auftretenden Polstellen der Funktion schon problematisch ist.

```
f1x = 'x/(4-sin(x))';
f2x = '4/tan(x)-x';

fplot(f1x,[0 20])
title('Pretty Plot  f1 = x/(4-sin(x)), x>0')
print ser3gr7.ps -dps
```

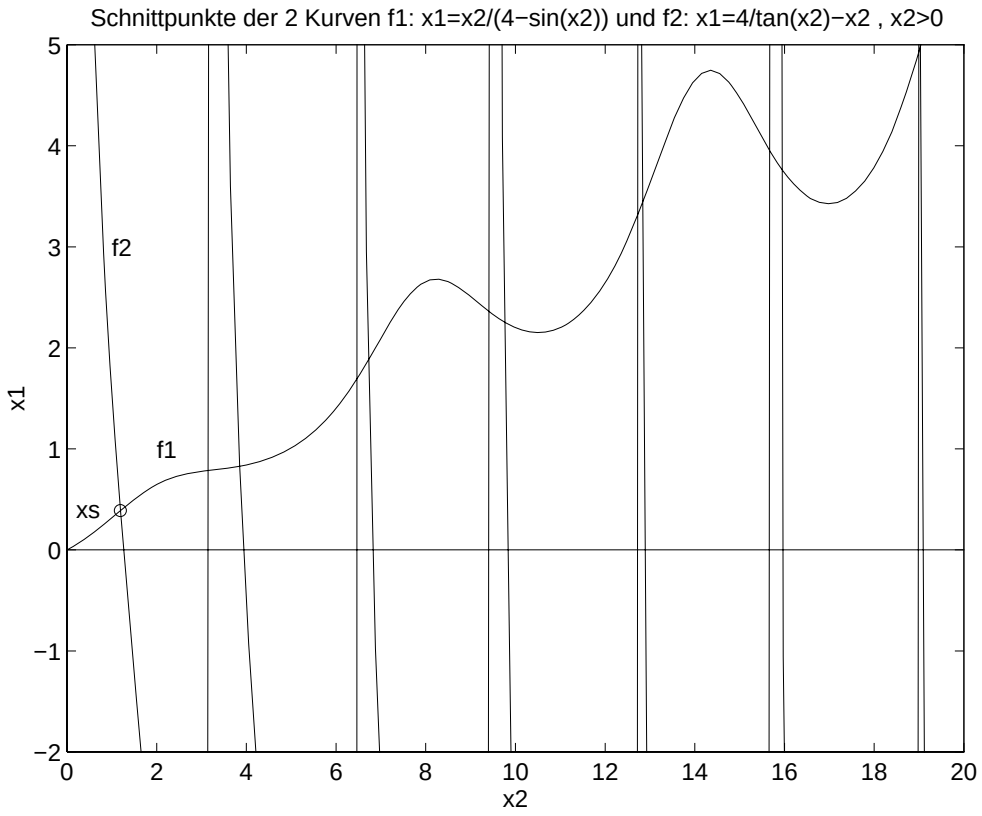
```
fplot(f2x,[0 20])
print ser3gr8.ps -dps
title('Pretty Plot  f2 = 4/tan(x)-x, x>0')
```




Beide Kurven zusammen ergeben, daß der gerade berechnete Fixpunkt nahe dem Koordinatenursprung zu erkennen ist. Das System hat unendlich viele Fixpunkte.

```
x = linspace(0.01,20.01,100);
for k=1:max(size(x))
  y1(k) = numeric(subs(f1x,num2str(x(k)),'x'));
  y2(k) = numeric(subs(f2x,num2str(x(k)),'x'));
end;
plot(x,y1,'y',x,y2,'g',x,0*y1,'w',1.19,0.39,'o')
axis([0 20 -2 5])
xlabel('x2')
ylabel('x1')
title('Schnittpunkte von f1: x1=x2/(4-sin(x2)) und
                         f2: x1=4/tan(x2)-x2 , x2>0')
text(2,1,'f1')
text(1,3,'f2')
text(0.2,0.4,'xs')
print ser3gr9.ps -dps

disp('1.Ableitungen')
f1xs = diff(f1x,'x')

f1xs =
1/(4-sin(x))+x/(4-sin(x))^2*cos(x)

f2xs = diff(f2x,'x')

f2xs =
-4/tan(x)^2*(1+tan(x)^2)-1
```



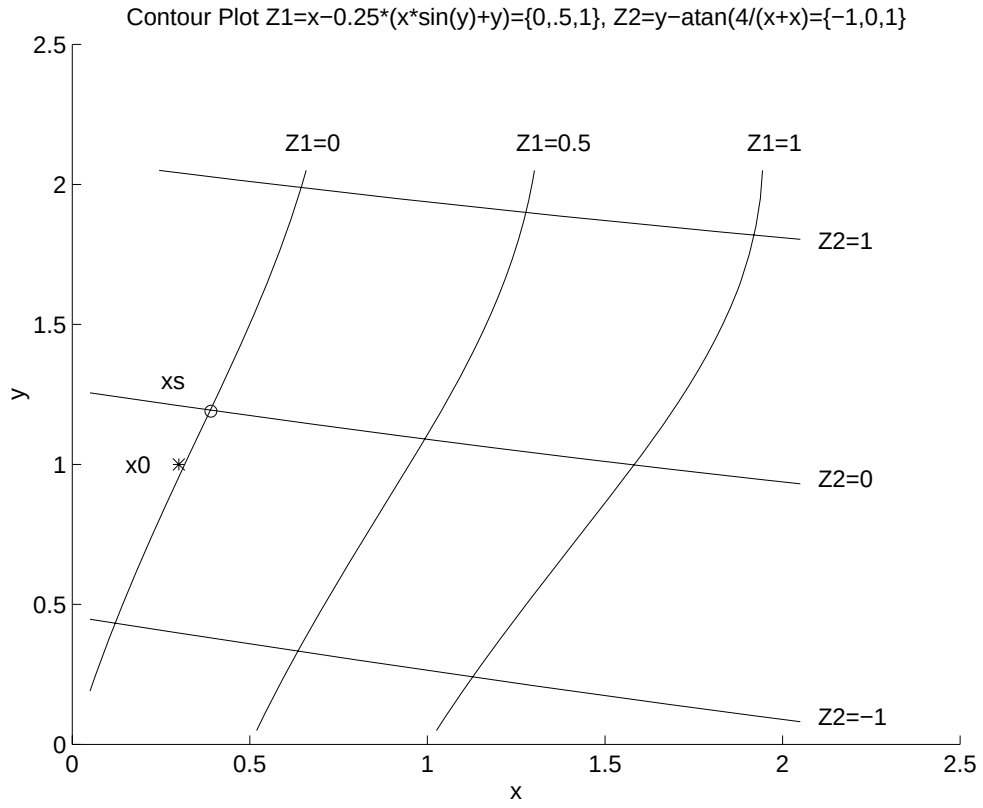

Um sich mehr Klarheit über die Funktionen zu verschaffen, bieten sich zahlreiche graphische Kommandos mit Plot von Flächen und Konturen an. Basis sind die abgeleiteten Funktionen

$$\begin{aligned} z_1(x,y) &= x - \frac{1}{4}(x \sin y + y), \\ z_2(x,y) &= y - \arctan \frac{4}{x+y}. \end{aligned}$$

Eine Übersicht sowie den Kontur-Plot im Bereich $(x,y) \in [0.2] \times [0,2]$ mit den entscheidenden Höhenlinien "0" erzeugen wir wie folgt.

```
% Teil 3  Plots
xx = 0.05:0.1:2.05;
yy = 0.05:0.1:2.05;
[X,Y] = meshgrid(xx,yy);
Z1 = X-0.25*(X.*sin(Y)+Y);
Z2 = Y-atan(4./(X+Y));

mesh(X,Y,Z1)
surf(X,Y,Z1)
contour(xx,yy,Z1,10)   % 10 Hoehenlinien
ve=0:1:1;
contour(xx,yy,Z1,ve)   % Hoehenlinien 0,1
% analog Z2
```

```
% Hoehenlinien mit Start- und Fixpunkt

ve1 = 0:0.5:1;
ve2 = -1:1:1;
hold on
contour(xx,yy,Z1,ve1)
contour(xx,yy,Z2,ve2)
title('Contour Plot Z1=x-0.25*(x*sin(y)+y)={0,.5,1},
                    Z2=y-atan(4/(x+x)={-1,0,1}')
xlabel('x')
ylabel('y')
plot(0.39,1.19,'o',0.3,1,'*')
text(0.25,1.3,'xs')
text(0.15,1.0,'x0')
text(0.6,2.15,'Z1=0')
text(1.25,2.15,'Z1=0.5')
text(1.9,2.15,'Z1=1')
text(2.1,1.8,'Z2=1')
text(2.1,0.95,'Z2=0')
text(2.1,0.1,'Z2=-1')
hold off
print ser3gr10.ps -dps
```



Grundlage der Konvergenzuntersuchungen ist die Jacobimatrix der rechten Seite $g$ der Fixpunktgleichung.

$$J(x) = g'(x) = \left(\frac{\partial g_i}{\partial x_j}\right) = \begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} \end{pmatrix} = \begin{pmatrix} \frac{1}{4}\sin x_2 & \frac{1}{4}(1 + x_1 \cos x_2) \\ -\frac{4}{16 + (x_1 + x_2)^2} & -\frac{4}{16 + (x_1 + x_2)^2} \end{pmatrix}$$

Es gilt für den Startpunkt

$$J(x^{(0)}) = \begin{pmatrix} 0.2104 & 0.2905 \\ -0.2261 & -0.2261 \end{pmatrix}, \quad \|J(x^{(0)})\|_\infty = 0.5009, \quad \|J(x^{(0)})\|_1 = 0.5166 < 1.$$

### Ausführung in MATLAB

```
% Teil 4  Jacobimatrix von g, siehe auch "jacobian"
g1 = '0.25*(x1*sin(x2)+x2)';
g2 = 'atan(4/(x1+x2))';
g11 = diff(g1,'x1')

g11 =
.25*sin(x2)

g12 = diff(g1,'x2')

g12 =
.25*x1*cos(x2)+.25

g21 = diff(g2,'x1')

g21 =
-4/(x1+x2)^2/(1+16/(x1+x2)^2)

g22 = diff(g2,'x2')

g22 =
-4/(x1+x2)^2/(1+16/(x1+x2)^2)

% Kontrolle g1(1,2)
hg11 = subs(g1,'1','x1');   % = .25*sin(x2)+.25*x2
hg12=subs(hg11,'2','x2');   % = .25*sin(2)+.50

numeric(hg12)

ans =
    0.7273
```

```
% Jacobimatrix J , symbolische Definition
% nicht nutzbar ist
Jacobi = sym('[g11,g12; g21,g22]')

Jacobi =
[ g11,g12]
[ g21,g22]

% sinnvoll mit Kommando  jacobian
w = sym('0.25*(x1*sin(x2)+x2); atan(4/(x1+x2))')

w =
[0.25*(x1*sin(x2)+x2)]
[     atan(4/(x1+x2))]

v = sym('x1,x2')

v =
[x1,x2]

Jacobi = jacobian(w,v)

Jacobi =
[                   .25*sin(x2),             .25*x1*cos(x2)+.25]
[-4/(x1+x2)^2/(1+16/(x1+x2)^2), -4/(x1+x2)^2/(1+16/(x1+x2)^2)]

% Kontrolle der Konvergenzbedingung am Startpunkt
Ja1 = subs(Jacobi,'0.3','x1')

Ja1 =
[                   .25*sin(x2),             .75e-1*cos(x2)+.25]
[-4/(.3+x2)^2/(1+16/(.3+x2)^2), -4/(.3+x2)^2/(1+16/(.3+x2)^2)]

Ja2 = subs(Ja1,'1.0','x2')

Ja2 =
[      .25*sin(1.0), .75e-1*cos(1.0)+.25]
[-.2261164499717355,  -.2261164499717355]

Ja3 = numeric(Ja2)

Ja2 =
    0.2104    0.2905
   -0.2261   -0.2261
```

```
lambdai = norm(Ja2,inf)

lambdai =
    0.5009

lambda1 = norm(Ja2,1)

lambda1 =
    0.5166
```

### Fehlerabschätzungen

```
% Teil 5  Erste Naeherungen mit AIV und Abschaetzung

x0 = [0.3 1.0]';
xk = x0;
xk1 = x0;
xxn = zeros(2,9);
xxn(:,1) = x0;
disp('AIV, 8 Iterationen')
disp('k               xk')
disp('---------------------------------')
for k=1:8
  hg11=subs(g1,xk(1),'x1');
  hg12=subs(hg11,xk(2),'x2');
  xk1(1) = numeric(hg12);
  hg21=subs(g2,xk(1),'x1');
  hg22=subs(hg21,xk(2),'x2');
  xk1(2) = numeric(hg22);
  xxn(:,k+1) = xk1;
  xk = xk1;
  s1 = num2str(k);
  s2 = sprintf('%11.9f',xk(1));
  s3 = sprintf('%11.9f',xk(2));
  disp([s1,'  ',s2,' ',s3])
end;

format long
xexakt = [0.388899948773 1.193984154565]'
format short
disp('a posteriori-Abschaetzung')
w = lambdai/(1-lambdai)*norm(xxn(:,9)-xxn(:,8),inf);
sprintf('|x8-x*| = %11.9f <= %11.9f = lambdai/(1-lambdai)|x8-x7|',...
         norm(xxn(:,9)-xexakt,inf),w)
```

```
AIV, 8 Iterationen
k               xk
---------------------------------
1  0.313110324 1.256564428
2  0.388585753 1.196842600
3  0.389643330 1.193434330
4  0.388915579 1.193942317
5  0.388891626 1.193989819
6  0.388899633 1.193984729
7  0.388900040 1.193984099
8  0.388899954 1.193984147

xexakt =
   0.38889994877300
   1.19398415456500

a posteriori-Abschaetzung

ans =
|x8-x*| = 0.000000008 <= 0.000000086 = lambdai/(1-lambdai)|x8-x7|
```

## 6.4 Lösung einer komplexen Gleichung

Bestimme die komplexe Lösung der Gleichung $e^z = z$, $z = x + iy$,
mit der Startnäherung $z_0 = 0.2 + i1.1$. Untersuche die Lösungsmenge der Gleichung.
Ein Versuch mit dem Kommando `fzero` zeigt uns, das dieses nur auf reelle Funktionen anwendbar ist.

```
f = 'exp(x)-x';
z0 = 0.2+1.1*i

z0 =
   0.2000 + 1.1000i

% fzero(f,z0)  nicht mit komplexen Zahlen
```

Also wandeln wir die Gleichung durch den Vergleich von Real- und Imaginärteil auf beiden Seiten in ein nichtlineares System mit zwei Unbekannten um.

$$\begin{aligned} f_1(x,y) &= x - e^x \cos y = 0, \\ f_2(x,y) &= y - e^x \sin y = 0. \end{aligned}$$

Daraus ist auch sofort eine Fixpunktvariante ablesbar. Jedoch wird die Fixpunktiteration wahrscheinlich kein akzeptables Konvergenzverhalten aufweisen, weil die großen Beträge der partiellen Ableitungen von $e^x \cos y$ bzw. $e^x \sin y$ bei $x \gg 1$ stören.

Wir wenden uns deshalb gleich dem modifizierten NV zu und nutzen die Funktion *newtons*.

```
% f7.m
function y = f7(x)
  y = zeros(1:max(size(x)))';
  y(1) = x(1) - cos(x(2))*exp(x(1));
  y(2) = x(2) - sin(x(2))*exp(x(1));

% Modifiziertes NV  mit m-File newtons.m
x0 = [0.2 1.1]';
f7x0 = f7(x0)

f7x0 =
   -0.3540
    0.0115

[xs,Defectx,Defectf,Error1,ItNr] = newtons(x0,'f7',1e-6,1e-6,50);

disp('Ergebnisse ');
disp(['xs(1,2)      = (',sprintf('%9.6f',xs(1)),', ',...
       sprintf('%9.6f',xs(2)),')']);
disp(['Defectx(1,2) = (',sprintf('%9.6e',Defectx(1)),', ',...
       sprintf('%9.6e',Defectx(2)),')']);       % (|xnp1-xn|,|ynp1-yn|)
disp(['Defectf      = ',sprintf('%9.6e',Defectf)]);  % ||f(x,y)||
disp(['Error1       = ',Error1]);
s1 = num2str(ItNr);
disp(['ItNr         = ',s1]);

Ergebnisse
xs(1,2)      = ( 0.318131,  1.337236)
Defectx(1,2) = (3.644797e-004, 5.295159e-004)
Defectf      = 2.861883e-007
Error1       = Small norm value of function
ItNr         = 3
```

An dieser Stelle ein kurze Diskussion zur Lösungssuche und -menge.

Man kann jede der Funktionen $f_1$ und $f_2$ einzeln untersuchen, insbesondere ihre Höhenlinien. Aus beiden Gleichungen folgt unmittelbar $\tan y = \frac{y}{x}$, $x = y/\tan y$, und wegen $e^{2x}(\cos^2 y + \sin^2 y) = x^2 + y^2$, $y^2 = e^{2x} - x^2$, erhalten wir für $y$ die nichtlineare Gleichung

$$y^2(1 + \tan^{-2} y) - e^{\frac{2y}{\tan y}} = 0.$$

(0,0) ist keine Lösung. Es gibt aber unendlich viele Lösungspunkte. Wenn $(x^*, y^*)$ Lösung ist, dann gilt stets $x^* > 0$ und $(x^*, -y^*)$ ist auch Lösung (Symmetrie zu $x$-Achse). Letzteres ergibt sich aus den Beziehungen

$$\begin{aligned} x - e^x \cos y &= x - e^x \cos(-y), \\ y - e^x \sin y &= y + e^x \sin(-y) = -[(-y) - e^x \sin(-y)]. \end{aligned}$$

Mit wachsendem $x^*$ wächst auch $|y^*|$ und $|y^*| \approx (2k + \frac{1}{2})\pi,\ k = 0, 1, 2, ...$

**Übersicht zum Konvergenzverhalten für verschiedene Startpunkte.**
Toleranzen und maximale Iterationszahl wie oben. Ende der Iteration stets mit
`Small norm value of function` oder `No further change of solution`,
$f_i^{(k)} = f_i(x^{(k)}, y^{(k)}),\ i = 1, 2.$

| $(x^{(0)}, y^{(0)})$ | $(f_1^{(0)}, f_2^{(0)})$ | $k$ | $(x^{(k)}, y^{(k)})$ | $(\|x^{(k)} - x^{(k-1)}\|, \|y^{(k)} - y^{(k-1)}\|)$ | $\|\|f(x^{(k)}, y^{(k)})\|\|_2$ |
|---|---|---|---|---|---|
| ( 0.2, 1.1) | (-0.3540, 0.0115) | 3 | ( 0.318131, 1.337236) | (3.644797e-004, 5.295159e-004) | 2.861883e-007 |
| ( 0.2,-1.1) | (-0.3540,-0.0115) | 3 | ( 0.318131,-1.337236) | (3.643225e-004, 5.293191e-004) | 2.833119e-007 |
| ( 2.0, 7.5) | (-0.5613, 0.5691) | 3 | ( 2.062278, 7.588631) | (1.868333e-005, 6.966153e-006) | 1.251927e-009 |
| ( 2.7,14.0) | ( 0.6654,-0.7400) | 3 | ( 2.653192,13.949208) | (2.830056e-006, 2.137270e-007) | 1.597894e-010 |
| ( 3.0,20.0) | (-5.1965, 1.6630) | 4 | ( 3.020240,20.272457) | (2.397521e-007, 1.497917e-007) | 5.759423e-006 |
| ( 2.0,20.0) | (-1.0153,13.2542) | 6 | ( 3.020240,20.272458) | (2.298573e-007, 7.892263e-009) | 4.685639e-006 |
| ( 1.0,20.0) | (-0.1093,17.5184) | 10 | ( 3.020240,20.272458) | (1.925084e-005, 3.106091e-006) | 5.492134e-009 |
| ( 0.0,20.0) | (-0.4081,19.0871) | 25 | ( 0.318132, 1.337236) | (1.319243e-005, 8.812252e-006) | 1.865988e-010 |
| ( -1.0,20.0) | (-1.1501,19.6641) | 14 | ( 0.318132, 1.337235) | (7.224820e-004, 2.214234e-004) | 3.869010e-007 |

## 6.5 Weiteres Beispiel

### Modifiziertes NV

```
% fv.m
function y = fv(x)
  y = zeros(1:max(size(x)))';
  y(1) = x(1) - 0.25*(x(1)*sin(x(2))+x(2));
  y(2) = x(2) - atan(4/(x(1)+x(2)));

disp('Newton-Verfahren fuer Systeme')
disp('****************************')
x0 = [0.3,1.0]';
[xs,Defect,Error1,fCalls] = newtons1(x0,'fv',1e-12,20)

xs =
    0.3889
    1.1940

Defect =
    0.2569

Error1 =
No further change of solution

fCalls =
    13
```

# 7 Anhang

## Anhang A

## Zusammenstellung von Adressen

### 1. World Wide Web (WWW) mit MATLAB Seiten

Die TEX Quelle sowie das PostScript file `primer35.ps` der 2.Edition des MATLAB Primers steht stehen mittels *ftp* auf `ftp.math.ufl.edu` im Verzeichnis `pub/matlab` zur Verfügung.

Die MathWorks Inc. und MathTools Ltd. entwickeln und vertreiben die Computersoftware MATLAB und andere Komponenten und sind natürlich mit ihrem ganzen Angebot auch im Internet zu finden.

| | |
|---|---|
| http://www.mathworks.com/ | The MathWorks Inc. home (auch Simulink) |
| http://www.mathworks.com/products/matlab/ | MATLAB 5.3 |
| http://www.mathworks.com/products/matlab/ | MATLAB web server 1.0 |
| http://www.mathworks.com/support/books/ | MATLAB based books |
| http://www.mathtools.com/ | MATLAB Toolboxen von MathTools Ltd. (auch MATCOM, MIDEVA) |
| http://krum.rz.uni-mannheim.de/cafgbench.html | Computer Algebra Benchmarks (RZ/Uni Mannheim) |

### 2. Verzeichnisse mit MATLAB *m*-Files für Skripte und Funktionen

Zu den Kapiteln 1-6 des Skripts liegen die entsprechenden Files (meist *m*-Files) und diverse Daten- und Ergebnisfiles vor.

`*.m, *.ps, *.eps, *.mat, *.txt, *.dat`

**Die Dateien sind zu finden im Novell-Netz PIVOT des Instituts für Mathematik bzw. auf der persönlichen Homepage im Internet.**

**\\PIVOT\SHARE Q:\NEUNDORF\STUD_M93\MATLAB2**

**Homepage Navigator → Publications → Computeralgebra → MATLAB2**

**e-mail:** ***neundorf@mathematik.tu-ilmenau.de***

**Homepage:** ***http://imath.mathematik.tu-ilmenau.de/~neundorf/index_de.html***

# Literatur

[1] Sigmon, K.: *MATLAB Primer, Second Edition.*
Department of Mathematics, University of Florida USA 1992.

[2] Sigmon, K.: *MATLAB Primer 5e.* CRC Press 1998.

[3] *MATLAB User's Guide and Reference Guide.*

[4] *MATLAB Release Notes 4.1. For Unix Workstations.*
The MathWorks Inc. Natick, Massachusetts USA 1993.

[5] *The Student Edition of MATLAB. Version 4. User's Guide.*
The MathWorks Inc. Prentice Hall, Englewood Cliffs New Jersey 1995.

[6] *The Student Edition of MATLAB 5.* Prentice Hall.

[7] Redfern, D.; Campbell, C.: *The MATLAB 5 Handbook.* Springer-Verlag 1998.

[8] Köckler, N.: *Numerische Algorithmen in Softwaressystemen: unter besonderer Berücksichtigung der NAG-Bibliothek.* B.G. Teubner Stuttgart 1990.

[9] Neundorf, W.: *MATLAB - Teil I: - Vektoren, Matrizen, lineare Gleichungssysteme.* Preprint M 20/99 IfMath der TU Ilmenau, Juli 1999.

[10] Mathews, J.H.; Fink, K.D.: *Numerical Methods using MATLAB.* Prentice Hall London 1999.

[11] Chen, K.; Giblin, P.J.; Irving, A.: *Mathematical explorations with MATLAB.* Cambridge University Press 1999.

[12] Mohr, R.: *Numerische Methoden in der Technik: eine Lehrbuch mit MATLAB-Routinen.* Vieweg Braunschweig 1998.

[13] Golubitsky, M.; Dellnitz, M.: *Linear algebra and differential equations using MATLAB.* Brooks/Cole Pub. Co Pacific Grove 1999.

[14] Kiełbasiński, A.; Schwetlick, H.: *Numerische lineare Algebra. Eine computerorientierte Einführung.* VEB Deutscher Verlag der Wissenschaften Berlin 1988.

## Anschrift:

Dr. Werner Neundorf
Technische Universität Ilmenau, Institut für Mathematik
PF 10 0565
D - 98684 Ilmenau

e-mail : neundorf@mathematik.tu-ilmenau.de